



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

DISSERTATION

**AUTOMATED BATTLE PLANNING FOR COMBAT
MODELS WITH MANEUVER AND FIRE SUPPORT**

by

Byron R. Harder

March 2017

Dissertation Supervisor

Christian Darken

This dissertation was performed at the MOVES Institute.

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE March 2017		3. REPORT TYPE AND DATES COVERED Dissertation
4. TITLE AND SUBTITLE AUTOMATED BATTLE PLANNING FOR COMBAT MODELS WITH MANEUVER AND FIRE SUPPORT			5. FUNDING NUMBERS RVL3T RVK4V	
6. AUTHOR(S) Byron R. Harder				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) Production combat models and simulations do not have an automated planning capability for multiple units in a command structure, and they lack tools for forward reasoning about effects such as suppression by firepower. These gaps limit modeled battle planning to the speed of meticulous human input. We present an architectural framework for automated battle planners, advocating a separation-of-duties approach—for example, between maneuver and fires—for the design and management of complex planning systems. We then describe a conceptual model for an automated fire support planning component, using greedy best-first search in continuous-time plan-space to reduce the risk of a given maneuver plan in polynomial time. We present an implementation of this component in an architecture conformant to the planning framework. We then describe a quantitative and qualitative approach to verification and validation of a planning model, and apply it to the fire support planner implementation. The results demonstrate that the automated fire support plans are effective at improving simulated combat results after a reasonable running time and have some realistic emergent properties. In addition to the novel planning algorithm, this research provides design principles, evaluation techniques, and promising results to guide improvements in behavior automation for combat models.				
14. SUBJECT TERMS modeling and simulation, automated planning, cognitive behavior, artificial intelligence, combat models, verification and validation, architectures, planning framework, derived models, risk intervals, fire support, hierarchical task networks			15. NUMBER OF PAGES 477	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

**AUTOMATED BATTLE PLANNING FOR COMBAT MODELS WITH
MANEUVER AND FIRE SUPPORT**

Byron R. Harder
Lieutenant Colonel, United States Marine Corps
M.S., Naval Postgraduate School, 2008
B.S., Vanderbilt University, 1999

Submitted in partial fulfillment of the
requirements for the degree of

**DOCTOR OF PHILOSOPHY IN MODELING, VIRTUAL ENVIRONMENTS,
AND SIMULATION**

from the

**NAVAL POSTGRADUATE SCHOOL
March 2017**

Approved by:	Christian Darken Professor of Computer Science Dissertation Supervisor	Imre Balogh Director, MOVES Institute Dissertation Committee Chair
	Neil Rowe Professor of Computer Science	Jeffrey Appleget Professor of Operations Research
	Craig Whiteside Professor of National Security Affairs	

Approved by: Peter Denning, Chair, Department of Computer Science

Approved by: Douglas Moses, Vice Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Production combat models and simulations do not have an automated planning capability for multiple units in a command structure, and they lack tools for forward reasoning about effects such as suppression by firepower. These gaps limit modeled battle planning to the speed of meticulous human input. We present an architectural framework for automated battle planners, advocating a separation-of-duties approach—for example, between maneuver and fires—for the design and management of complex planning systems. We then describe a conceptual model for an automated fire support planning component, using greedy best-first search in continuous-time plan-space to reduce the risk of a given maneuver plan in polynomial time. We present an implementation of this component in an architecture conformant to the planning framework. We then describe a quantitative and qualitative approach to verification and validation of a planning model, and apply it to the fire support planner implementation. The results demonstrate that the automated fire support plans are effective at improving simulated combat results after a reasonable running time and have some realistic emergent properties. In addition to the novel planning algorithm, this research provides design principles, evaluation techniques, and promising results to guide improvements in behavior automation for combat models.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	OVERVIEW.....	1
B.	ORGANIZATION OF DOCUMENT.....	1
C.	BACKGROUND.....	2
	1. Combat Modeling and Simulation.....	2
	2. Battle Planning.....	4
	3. Human Behavior in Combat Simulations.....	6
	4. Current Processes for M&S Battle Planning.....	7
	5. Fire Support Planning.....	9
D.	PROBLEM DISCUSSION.....	12
	1. Limitations to Analysis.....	12
	2. Automated Replanning.....	13
	3. Training Implications.....	13
	4. The Foundational Role of Initial Planning.....	14
	5. Formalizing the Planning Problem.....	14
E.	CONTRIBUTIONS.....	16
F.	NEED FOR STUDY.....	18
G.	CHALLENGES AND POTENTIAL IMPACT.....	22
II.	LITERATURE AND TECHNOLOGY REVIEW.....	25
A.	OVERVIEW.....	25
B.	TACTICS.....	25
	1. Tactical Considerations for an Infantry Attack.....	25
	2. Integrating Fire Support.....	26
C.	AUTOMATED PLANNING FUNDAMENTALS.....	29
	1. Generic Framework for Planning.....	30
	2. Classical Planning.....	31
	3. Plan-Space Planning.....	33
	4. Hierarchical Task Networks.....	35
	5. State-Variable Representation.....	38
	6. Continuous-Time Planning.....	39
	7. Planning as Search.....	43
	8. Evolutionary Planning.....	45
	9. Extensions to Plan Representations.....	46
D.	RELATED TOPICS IN BEHAVIOR DEVELOPMENT.....	47
	1. Movement and Pathfinding.....	47
	2. Behavior Trees.....	50

3.	Entity and Small Unit Movement	52
4.	Cognitive Architectures	53
5.	Combat Outcome Prediction	54
6.	Fire Support in Combat Models	58
7.	Human Behavior Validation	58
E.	RELEVANT MODELS	61
1.	COMBATXXI	61
2.	One Semi-Automated Forces	66
3.	Virtual Battlespace	68
4.	Marine Air Ground Task Force Tactical Warfare Simulation	70
5.	Map Aware Non-uniform Automata-Vector	71
6.	RTS Games	72
F.	PLANNING IN M&S AND GAMES	75
1.	Goal-Oriented Action Planning	75
2.	HTN Applications	77
3.	Planned Assault	78
4.	Cover State Graph Planning	80
5.	Command Ops 2	83
6.	Evolutionary Battle Planners	85
G.	SUMMARY	87
III.	CONCEPTUAL PLANNING FRAMEWORK	91
A.	MOTIVATION AND ORGANIZATION	91
1.	Top-Level Framework	92
2.	Other Useful Terms	93
3.	Separation of Duties	95
4.	Version Control	96
B.	PLANNING DATA	98
1.	Definitions and Concepts	99
2.	Derived Model Dictionary	105
3.	Task Dictionary	108
4.	Method Dictionary	109
5.	Standard Tactics Configuration Dictionary	110
C.	PLANNING INPUT	112
1.	Map	113
2.	Friendly Force	114
3.	Enemy Force	116
4.	Other Actors	118
5.	Tasking	119

	6.	Tactics	124
D.		PLAN GENERATOR.....	125
	1.	Input Processor	125
	2.	Planning Controller	126
	3.	Annotated Mobility Graph.....	126
	4.	Pathfinding	127
	5.	Partial Plans	127
	6.	Mission Planner.....	128
	7.	Enhancement Planner	131
	8.	Plan Compiler.....	132
	9.	Separate Planning Systems	132
E.		SUMMARY	134
IV.		THE FIRE SUPPORT PLANNER.....	137
A.		OVERVIEW	137
B.		ASSUMPTIONS AND INPUT.....	137
	1.	Time and Terrain.....	138
	2.	Units and Threat Annotations	138
	3.	Actions and Routes	139
	4.	Availability Tasks.....	141
	5.	Suppression.....	142
C.		DATA ELEMENTS	143
	1.	Risk Object Hierarchy.....	143
	2.	Fire Support Tasks	150
	3.	Planning Nodes.....	151
D.		OPERATIONS	151
	1.	Immediately Following Task.....	152
	2.	Route Replacement	152
	3.	Availability Task Remainders	153
	4.	Risk Subdivision.....	155
	5.	Risk Reduction	157
	6.	Task Scores and Affected Risk Sets	158
	7.	Unit Subsets	160
	8.	Planning Branch Generation and Removal.....	161
E.		SEARCH PROBLEM CHARACTERIZATION.....	161
	1.	Goal Test	162
	2.	Successor Function.....	163
	3.	Plan Cost	165
	4.	Search Strategy	167
F.		ALGORITHM.....	168

1.	Top-Level Procedure	169
2.	Choosing the Best Task	171
3.	Applying the Selected Task	171
4.	Generating Potential Fire Support Tasks	175
5.	Generating a Task Option for a Single Risk Interval	176
6.	Updating Affected Risk Sets	179
7.	Storing Affected Risk Sets	181
8.	Pathfinding	183
G.	COMPLEXITY ANALYSIS	185
1.	Expansion of Terms	186
2.	Additional Assumptions	189
3.	Results	190
4.	Summary of Complexity Analysis	193
H.	RISK VALUE CALCULATIONS	194
1.	Assumptions	194
2.	Risk Value for Risk Segments	194
3.	Risk Value for Risk Subregions	196
I.	EXTENSIONS	197
1.	Variable Unit Sizes	197
2.	Approximate Risk Values	198
3.	Attrition and Suppression Effects in the Maneuver Plan	199
4.	Target Selection by Threat Units	200
5.	Modifying Risk Values	203
6.	Lasting Suppression Effects and Area Fire	204
7.	Interacting Effects	204
8.	Realistic Unit Formations	205
V.	IMPLEMENTATION	207
A.	OVERVIEW	207
1.	Software Architecture	208
2.	Notation	210
B.	PLANNING DATA	211
1.	Unit Templates	211
2.	Derived Model Dictionary	211
3.	Task Dictionary	218
4.	Method Dictionary	222
5.	Standard Tactics Configuration Dictionary	226
C.	PLANNING INPUT	226
1.	Map	226
2.	Friendly Force	227

3.	Enemy Force.....	228
4.	Other Actors	229
5.	Tasking.....	229
6.	Tactics	230
D.	PLAN GENERATOR.....	231
1.	Input Processor	232
2.	Planning Controller	233
3.	Annotated Mobility Graph.....	233
4.	Pathfinding	240
5.	Partial Plans	247
6.	Risk Interval Data Structures.....	250
7.	Mission Planner.....	258
8.	Enhancement Planner	260
9.	Plan Compiler.....	269
10.	Separate Planning Systems	269
E.	SUMMARY	270
VI.	TESTING.....	271
A.	OVERALL TEST DESIGN	271
1.	Iterative Testing and Development Process	271
2.	Considerations for Verification, Validation, and Accreditation	273
B.	FUNCTIONAL TESTING.....	275
1.	Annotated Mobility Graph Visualization	275
2.	Geographic Task Visualization.....	277
3.	Hierarchical View of the Plan.....	279
4.	Textual Display of Risk Intervals and Fire Support Tasks....	281
5.	Geographic Error Visualization	284
6.	Summary of Functional Testing	285
C.	SCENARIOS FOR QUANTITATIVE AND QUALITATIVE TESTING.....	286
1.	Force Pairings.....	286
2.	Maps	289
3.	Planning Types	292
4.	Summary of Testing Scenarios	295
D.	QUANTITATIVE TESTING	296
1.	Testing Procedures.....	297
2.	Summary Results	300
3.	Analysis of Scalability Results	303
4.	Analysis of Simulated Combat Performance Results	307

E.	QUALITATIVE TESTING	317
1.	Well-Timed Suppression	318
2.	Shifting and Lifting Fires	319
3.	Supporting Follow-on Attacks	321
4.	Consolidated Fire Support Positions.....	322
5.	Bounding Movement.....	323
6.	Avoiding Threats.....	324
7.	Ignoring Low Threats.....	324
8.	Unsafe Directions of Fire.....	325
9.	Firing Position Collocation.....	326
10.	Incorrect Cease-Fire Times.....	327
11.	Example Plan Output	328
12.	Summary of Qualitative Results.....	335
F.	SUMMARY OF TESTING RESULTS.....	336
VII.	CONCLUSION	337
A.	SUMMARY OF APPROACH.....	337
B.	DISCUSSION OF RESULTS	338
1.	The Case for the Conceptual Planning Framework	338
2.	The Fire Support Planner and Its Components.....	339
3.	Implementation	342
4.	Evaluating the Implementation	343
5.	Implications	344
C.	FUTURE WORK.....	348
APPENDIX	A. COMBAT SIMULATION ENVIRONMENT	
	IMPLEMENTATION	351
A.	SYSTEM PURPOSE AND PRINCIPLES.....	351
B.	DISCRETE EVENT SYSTEM.....	352
C.	TERRAIN	354
D.	ENTITIES.....	355
1.	Mover Manager.....	358
2.	Seeker	359
3.	TargetHandler	359
4.	TargetSensor	360
5.	Weapon	361
6.	EntityStatus	363
7.	EntityInformation	364
8.	UnitMember	366
E.	UNITS, COMMANDS, AND PLANS	366

F.	FORMATIONS	368
G.	TACTICAL CONTROL MEASURES	370
H.	ACTIONS	371
I.	RUN MANAGERS AND DATA COLLECTION	372
APPENDIX B. AUTOMATED BATTLE PLANNING SYSTEM DETAILS		377
A.	OVERVIEW	377
B.	VISIBILITY GRAPH EDITOR	377
C.	TASK NODE FUNCTIONS AND FIELDS	378
D.	PARTIAL PLANS	380
E.	METHODS	381
F.	RISK INTERVAL DATA STRUCTURES	382
1.	Risk Sets	382
2.	Risk Intervals	382
3.	Risk Subregions	384
G.	CONSTRUCTING THE MANEUVER PLAN	385
H.	POTENTIAL FIRE SUPPORT SETS	386
I.	PLANNING-RELATED OPERATIONAL TASK FEATURES	387
J.	FIRE SUPPORT TASKS	387
APPENDIX C. MILITARY OPERATIONS AND PLANNING		389
A.	OVERVIEW	389
B.	TACTICS	389
C.	ORGANIZATION OF FORCES	391
D.	LEVELS OF WARFARE	392
E.	OPERATIONS ORDERS AND PLANS	394
1.	Situation	395
2.	Mission	395
3.	Execution	395
4.	Administration and Logistics	396
5.	Command and Signal	396
F.	MILITARY PLANNING PROCESSES	396
1.	The Troop Leading Steps	397
2.	The Formal Planning Process	397
3.	Modeling the Planning Process	399
G.	TYPES OF OPERATIONS	400
H.	PHASES OF ATTACK OPERATIONS	401
I.	MOVEMENT AND FORMATIONS	402
J.	FORMS OF MANEUVER	404

APPENDIX D. TESTING SCENARIO DESCRIPTIONS	409
A. INTRODUCTION.....	409
B. SCENARIO GROUP 1: PLATOON VERSUS SQUAD, MAP A	409
C. SCENARIO GROUP 2: PLATOON VERSUS SQUAD, MAP B	413
D. SCENARIO GROUP 3: COMPANY VERSUS PLATOON, MAP B.....	417
E. SCENARIO GROUP 4: PLATOON VERSUS SQUAD, MAP C	421
F. SCENARIO GROUP 5: COMPANY VERSUS PLATOON, MAP C.....	424
G. SCENARIO GROUP 6: BATTALION VERSUS COMPANY, MAP C.....	428
 LIST OF REFERENCES	 435
 INITIAL DISTRIBUTION LIST	 445

LIST OF FIGURES

Figure 1.	Suppressive Fires in Support of an Assault. Adapted from FM 3–21.11 (DOD 2003).	11
Figure 2.	Generic Planning System Framework. Adapted from Ghallab, Nau, and Traverso (2004, 8–9).	30
Figure 3.	An HTN Decomposition Tree. Source: Ghallab, Nau, and Traverso (2004).	38
Figure 4.	A Behavior Tree. Source: Isla (2005).	51
Figure 5.	Human Behavior Representation Framework. Source: DOD (2001).	60
Figure 6.	CXXI Command Hierarchy. Source: DOD (2015a).	62
Figure 7.	A CXXI HTN in Behavior Studio. Source: MOVES Institute (2015).	64
Figure 8.	OneSAF Support By Fire Task Example. Source: DOD (2015c).	68
Figure 9.	VBS3 Waypoints. Source: Bohemia Interactive Simulations (2015).	69
Figure 10.	VBS3 Suppress Area Command. Source: Bohemia Interactive Simulations (2015).	70
Figure 11.	Four RTS Control Architectures. Adapted from Ontañón et al. (2013).	75
Figure 12.	A Partial Plan in the PlannedAssault Representation. Source: van der Sterren (2013).	79
Figure 13.	Suppression Targeting Example	82
Figure 14.	CO2 Hierarchical Order of Battle. Source: O’Connor (2015).	84
Figure 15.	CO2 Coordinated Attack. Source: O’Connor (2015).	85
Figure 16.	Top-level Framework.	93
Figure 17.	Planning Data.	98
Figure 18.	Planning Input	113
Figure 19.	The Plan Generator	125
Figure 20.	Separate Planning Systems	134
Figure 21.	A Unit Following a Route Along a Terrain Surface	141
Figure 22.	The Risk Object Hierarchy	143
Figure 23.	Threatened Portions of a Route.	148
Figure 24.	Route Replacement	152
Figure 25.	Availability Task Remainders.	154

Figure 26.	Risk Subdivision	156
Figure 27.	Increasing and Decreasing Risk Value, Part 1	166
Figure 28.	Increasing and Decreasing Plan Risk Value, Part 2.....	167
Figure 29.	Unequal Risk Segment Subdivision.....	199
Figure 30.	WXXI Automated Battle Planning System Class Diagram.....	209
Figure 31.	Column Formation With Threats	213
Figure 32.	Online Formation with Threats	214
Figure 33.	Friendly Force and Planner in the Unity Hierarchy Window	227
Figure 34.	Plan Generator Implementation	231
Figure 35.	Annotated Mobility Node	236
Figure 36.	Two Different Perspective Views of a Node	239
Figure 37.	An ABPath With No Modifiers Applied	245
Figure 38.	An ABPath with Radius and Funnel Modifiers Applied	245
Figure 39.	A Company Maneuver Plan Displayed in the Unity Transform Hierarchy.....	260
Figure 40.	Testing Phases and Feedback Loops.....	272
Figure 41.	Annotated Mobility Graph Visualization.....	276
Figure 42.	Movement Task Visualization	278
Figure 43.	Support By Fire Task Visualization.....	278
Figure 44.	Example of Suppression Effectiveness Evaluation.....	279
Figure 45.	Partial Plan Hierarchical View.....	281
Figure 46.	Example of Textual Debugging Output.....	283
Figure 47.	PlanFireSupport Score Verification Code Sample	284
Figure 48.	Example of Geographical Error Visualization.....	285
Figure 49.	Terrain A. Cayucos Creek, CA: 2 km ² , 8192 triangles.....	291
Figure 50.	Terrain B. Palo Alto, VA: 4 km ² , 32,768 triangles.....	291
Figure 51.	Terrain C. Buckeye Peak, CO: 8 km ² , 131,072 triangles	292
Figure 52.	Histograms of Number of Objectives Cleared for Platoon-versus- Squad Scenarios	309
Figure 53.	Histograms of Number of Objectives Cleared for Company-versus- Platoon Scenarios.....	309
Figure 54.	Histograms of Number of Objectives Cleared for Battalion-versus- Company Scenarios	310

Figure 55.	Histograms of FER for Platoon-versus-Squad Scenarios	313
Figure 56.	Histograms of FER for Company-versus-Platoon Scenarios.....	313
Figure 57.	Histograms of FER for Battalion-versus-Company Scenarios	313
Figure 58.	Box Plots of FER for All Scenarios	314
Figure 59.	Box Plots of ln(FER) for All Scenarios	315
Figure 60.	Box Plots (Less Outliers) of FER by Planning Type	317
Figure 61.	Two Machine Gun Squads Suppress the Objectives as Maneuver Units Begin Their Assault Movements.....	319
Figure 62.	Shifting and Lifting Behavior	320
Figure 63.	A Maneuver Unit Supporting a Follow-on Attack	321
Figure 64.	Emergent Company Fire Support Position	323
Figure 65.	Unsafe Direction of Fire	325
Figure 66.	Example Maneuver and Fire Support Plan Overhead View	329
Figure 67.	Fire Support Tasks for Machine Gun Squad 1.....	331
Figure 68.	Fire Support Tasks for Machine Gun Squad 2.....	332
Figure 69.	Fire Support Tasks for Machine Gun Squad 3.....	333
Figure 70.	Fire Support Tasks for Rifle Squads.....	334
Figure 71.	Fire Support Tasks for the Independent Rifle Squad.....	335
Figure 72.	Real World Terrain Helper. Source: Infinity Code (2016)	355
Figure 73.	WXXI Entity GameObject.....	357
Figure 74.	A Company Command Organization in the Transform Hierarchy.....	367
Figure 75.	Multiple Run Manager Inspector Pane	374
Figure 76.	Unity Build Settings.....	376
Figure 77.	Tactical Control Measures. Source: DOD (2012a).....	404
Figure 78.	Frontal Attack. Source: DOD (2014a).....	405
Figure 79.	Flanking Attack. Source: DOD (2014a).	406
Figure 80.	Scenario Group 1 Objective Area	410
Figure 81.	Scenario Group 1 Node Penalty Visualization	411
Figure 82.	Scenario Group 1 Maneuver Plan	412
Figure 83.	Scenario Group 1 Close-quarters Starting Positions.....	413
Figure 84.	Scenario Group 2 Objective Area	414
Figure 85.	Scenario Group 2 Node Penalty Visualization	415

Figure 86.	Scenario Group 2 Maneuver Plan	416
Figure 87.	Scenario Group 2 Close-quarters Starting Positions.....	417
Figure 88.	Scenario Group 3 Objective Area	418
Figure 89.	Scenario Group 3 Node Penalty Visualization	419
Figure 90.	Scenario Group 3 Maneuver Plan	420
Figure 91.	Scenario Group 3 Close-quarters Starting Positions.....	421
Figure 92.	Scenario Group 4 Objective Area	421
Figure 93.	Scenario Group 4 Node Penalty Visualization	422
Figure 94.	Scenario Group 4 Maneuver Plan	423
Figure 95.	Scenario Group 4 Close-quarters Starting Positions.....	424
Figure 96.	Scenario Group 5 Objective Area	425
Figure 97.	Scenario Group 5 Node Penalty Visualization	426
Figure 98.	Scenario Group 5 Maneuver Plan	427
Figure 99.	Scenario Group 5 Close-quarters Starting Positions.....	428
Figure 100.	Scenario Group 6 Objective Area	429
Figure 101.	Scenario Group 6 Node Penalty Visualization	430
Figure 102.	Scenario Group 6 Maneuver Plan Assaults	431
Figure 103.	Scenario Group 6 Starting Positions and Manual Fire Support Positions	432
Figure 104.	Scenario Group 6 Close-quarters Starting Positions.....	433

LIST OF TABLES

Table 1.	Documented Research Requirements.*	18
Table 2.	EFST for a Support by Fire Task	27
Table 3.	Testing Scenarios	296
Table 4.	Summary Quantitative Results	302
Table 5.	Preprocessing Scalability Results	304
Table 6.	Fire Support Planning Times	305
Table 7.	Simulated Combat Performance Results	308
Table 8.	Mission Accomplishment Score Detailed Results.....	312
Table 9.	Statistical Results for FER	316
Table 10.	Levels of Warfare and the Echelons within Them.....	393
Table 11.	Tactical Tasks and Forms of Maneuver by Echelon.....	407

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ABPS	Automated Battle Planning System
AI	Artificial Intelligence
BSL	Behavior Scripting Language
COMBAT XXI	Combined Arms Analysis Tool for the 21st Century
CSE	Combat Simulation Environment
CXXI	COMBAT XXI
DES	Discrete Event Simulation
DOD	Department of Defense
GOAP	Goal-Oriented Action Planning
HTN	Hierarchical Task Network
M&S	Modeling and Simulation
MACROP	Macro Operator
NPS	Naval Postgraduate Schools
RTS	Real-Time Strategy
S&T	Science and Technology
STN	Simple Task Network
STO	Science and Technology Objective
TRAC	TRADOC Analysis Command
TRADOC	U.S. Army Training and Doctrine Command
V&V	Verification and Validation
WXXI	Wombat XXI

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Today's combat simulation software is not capable of automatically generating realistic battle plans. Human scenario designers and operators perform the task instead. For Modeling and Simulation (M&S)-based analysis, a significant portion of the time required to complete a study is taken by the battle planning effort. This limits the number of scenarios for a study to only a few. For M&S-based training, the lack of an automated planning capability increases the requirement for human operators, limiting the efficiency gains that users seek from computer-based alternatives to live training. A few examples of automated maneuver planning have appeared in military and video game research, but so far, they have not seen use in production military simulations. No practical automated fire support planning capability has been demonstrated.

We present an architectural framework (Figure 1) for automated battle planning systems (ABPSs), advocating a separation-of-duties approach—for example, between maneuver and fires—for the design and management of complex planning systems. The major components of the framework are

- **Planning Data:** the tasks, methods (rules for achieving tasks), and derived models that lend characteristic features to plans
- **Planning Input:** the data taken from the combat simulation environment and user to formulate a planning problem for the ABPS
- **Plan Generator:** the algorithms, heuristics, data structures, and interfaces that actually produce battle plans

The framework is a natural way to delineate the responsibilities of three groups involved in automated battle planning: scenario designers, who configure the Planning Input for specific M&S problems; behavior developers, who author and update Planning Data to model different tactics; and automated planning developers, who work on the internals of the Plan Generator. The framework also introduces the *planning style*, which constrains units to an appropriate subset of tasks, methods, and derived models in order to control the options available from a Planning Data store that could grow quite large over time.

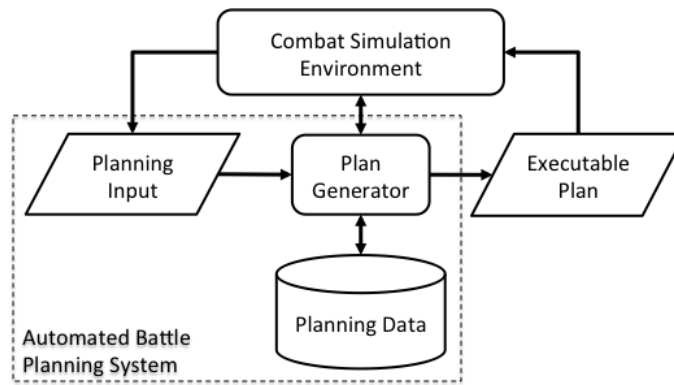


Figure 1. Conceptual Planning Framework

The Plan Generator includes a Plan Controller, which invokes multiple Mission Planner and Enhancement Planner subcomponents to produce partial plans. Mission Planners attempt to produce partial plans that qualitatively achieve the Required Tasks provided in the Planning Input, choosing less costly options when possible. Enhancement Planners expand or modify partial plans to improve them quantitatively according to one or more objective functions. Mission and Enhancement Planners approach battle planning, respectively, as problem solving and optimization. Each Mission or Enhancement Planner is focused on a particular aspect of planning, such as maneuver, fires, or sustainment. The Plan Generator ultimately compiles its internal plan representation into a format that can be executed in the target Combat Simulation Environment.

A few examples of maneuver planners have appeared in academic literature and industry—for example, William van der Sterren’s PlannedAssault system (www.plannedassault.com) for Virtual Battlespace (VBS). This type of tool could be used as a Mission Planner for an ABPS.

Modern military tactics rely on suppression of enemy threats by firepower. Although some combat models in current use include suppression effects, no automated planning tools exist to add fire support tasks to a maneuver plan. We present a conceptual model for a fire support planner, which can work as an Enhancement Planner for an ABPS. The fire support planner uses *risk intervals* (Figure 2) as its primary data elements. Each risk interval corresponds to a period of continuous time that a friendly

unit is exposed to an enemy unit’s potential fire. The numerical cost of each risk interval is the expected number of casualties the friendly unit would sustain while traversing the corresponding path. This value can be computed by integrating the threat’s killing rate over time. A *fire support task* is an instruction for a unit to move to a particular location and suppress a single threat unit for some period of time, temporarily reducing the threat’s killing rate. Each fire support task, once added to a plan, reduces the numerical cost of all risk intervals overlapped by its time interval.

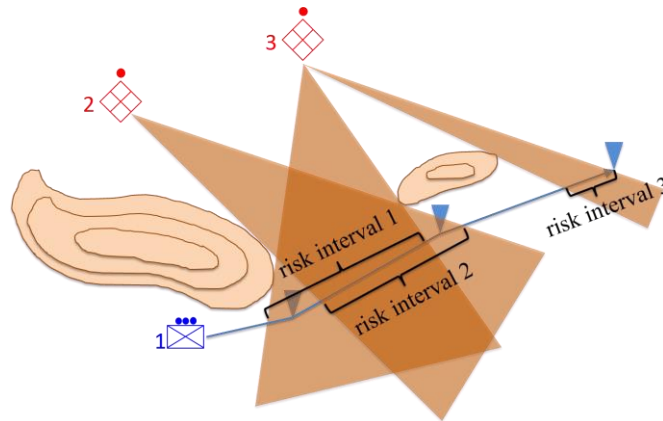


Figure 2. Risk Intervals

The plan cost for the fire support planner is defined as the sum of all of the plan’s risk interval costs, and the score of a candidate fire support task is its potential reduction to the plan’s cost if it were the next task added. We present a greedy best-first algorithm to construct a fire support plan based on this scheme. The fire support task options are built by attempting to apply every fire support asset to every risk interval, using multi-point tactical pathfinding to determine positions, routes, and timing. By using a plan-space representation, we are able to apply suppression to the most critical moments of the plan; we are not restricted to forward or backward chronological planning.

To implement the fire support planner, we first build a relatively simple combat model in the Unity 3D development platform called Wombat XXI. It includes a polygonal terrain elevation model imported from real world data, hierarchically organized units, formations for entity movement, range-dependent probability of hit, and

a suppression model similar to that of COMBATXXI, a production analytical combat simulation system. We construct a prototype ABPS in accordance with the aforementioned Conceptual Planning Framework. Its single Mission Planner builds maneuver plans for designated units in a hierarchical task network representation, making heavy use of user input. Its Enhancement Planner uses the fire support planner algorithm to add movement and suppression tasks to the plan, expending the untasked time intervals of capable units to reduce risk from enemy fire. The “manual” maneuver planner is useful in constructing specific test scenarios for the fire support planner.

We use a quantitative and qualitative approach for verification and validation of the fire support planner implementation. The process itself, shown in Figure 3, is reusable for future development efforts of a similar nature. Quantitative testing provides more robust evidence for the observations made during qualitative testing; qualitative testing checks whether the quantitative measures have a realistic and believable explanation.

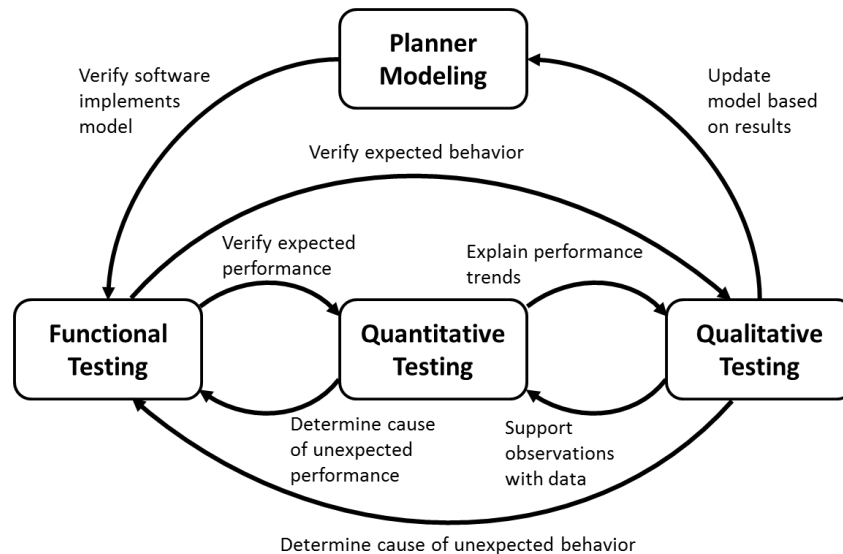


Figure 3. Verification and Validation Process

The quantitative results (Table 1) demonstrate that the automated fire support plans produce better simulated combat results than a simple human-designed plan, although we believe a human could eventually find better plans given unlimited time. The

practical running time ranges from less than a second for platoon-sized scenarios to a few minutes for a battalion-sized scenario. The algorithm appears scalable enough for online real-time applications of platoon and company size if run in a separate thread, and for offline planning it is significantly faster than manual human input. Our qualitative analysis reveals some interesting emergent properties in the automated plans that potential users should consider for validation. We also identify some shortcomings in the current version that an implementation would need to handle or accept.

Table 1. Quantitative Testing Results

Scenario				Running Time (sec)		Combat Performance		
Group	Forces	Map (km ²)	Fire Support Planning Type	Pre- processing	Planning	Mission Accomplishment	Fractional Exchange Ratio	
1	Plt vs Squad	2	Manual	0.07		36.7%	0.88	
			Automated		0.24	66.7%	5.13	
2	Plt vs Squad	4	Manual	0.28		56.7%	1.73	
			Automated		0.52	83.3%	19.50	
3	Co vs Plt		Manual	3.00		26.7%	0.40	
			Automated		4.39	73.3%	1.87	
4	Plt vs Squad	8	Manual	3.90		45.0%	0.53	
			Automated		0.58	87.5%	8.88	
5	Co vs Plt		Manual	7.91		46.0%	0.89	
			Automated		10.62	75.0%	3.17	
6	Bn vs Co		Manual	21.18		15.6%	0.38	
			Automated		187.56	84.1%	2.56	

In addition to a novel and effective fire support planning algorithm with a functioning implementation, this research provides design principles, evaluation techniques, and promising results to guide improvements in automated battle planning for combat models. The stage is set to bring this capability into production systems. Although it will require an upfront investment and continued maintenance and management, there is enough demonstrated technology to improve the efficiency of scenario design. By allowing careful human review of automated plans, we can build the trust of the modeling community while simultaneously refining the automation. Once a sufficient level of comfort and understanding is achieved, online replanning may revolutionize the way that combat models are used in a variety of different applications.

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

This dissertation is dedicated to my wife Grace, who has been unhesitating and unwavering in her support for my professional and academic goals. She set aside her own impressive career to be with me, entering a more unpredictable and often frustrating world than she could possibly have imagined. My pursuit of a doctoral degree asked even more of her—she often had to operate as a single parent as I spent all hours reading, thinking, coding, and writing. She sacrificed at least as much as I did during this process, and she never asked for any recognition beyond the love of her family.

Thanks to my dissertation committee for so much precious time and insight. They challenged me to extend the limits of my abilities in research, analysis, and writing. They gave me the latitude to explore new facets of the problem, and their guidance allowed me the most efficient use of my time at NPS. Chris Darken, my advisor, awoke my simmering interest in AI topics with his course on cognitive modeling. I considered every meeting and review with him a learning experience and a privilege. Imre Balogh kept my ideas moving forward with his behavior modeling seminar, and he showed me the importance of high-level “big ideas.” Neil Rowe is a brilliant researcher in a wide variety of topics. I knew that, since he was interested in my work, I must be on to something. He dedicated a significant amount of time to help keep me on track. Jeff Appleget is an accomplished leader in combat modeling, operations research, and wargaming. His input was invaluable, particularly with respect to verification and validation. Craig Whiteside, an infantry officer and thought leader on the contemporary security environment, kept me grounded in the real world and oversaw my review of tactical concepts and doctrine.

Thanks also to the Marines that have invested in greater technical education in our service. General Robert Neller, the Commandant of the Marine Corps, made the game-changing decision to pursue doctoral degrees for active duty Marines. LtGen Robert Walsh, Col Todd Lyons, LtCol Louis Camardo, Maj Harry Reifschneider, and Capt Ezra Akin all played important roles leading to that decision and supporting my efforts. Col Mitch McCarthy was my strongest advocate and best advisor through this entire program. I will strive, in his words, to “pay it forward.”

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. OVERVIEW

The objective of this dissertation is to improve the capability of automated battle planning systems (ABPSs), in particular those used for modeling and simulation (M&S). Our central contribution is the design of an automated fire support planner, which is conceived as a component of an automated battle planning system. To explain the architectural context of such a component, we first propose a Conceptual Planning Framework for ABPSs, which offers a new way to compartmentalize and bring together multiple, specialized ABPS-related technologies. The effectiveness of the fire support planner is evaluated through software implementation and testing. The general approach of the testing process itself can be applied to future ABPS components.

B. ORGANIZATION OF DOCUMENT

The remainder of this chapter presents background information to clarify the context of the problem and to provide definitions for terms that are new or have varied uses in the literature. It then discusses the problem in more detail and argues that there is a need for the solutions we are about to present.

Chapter II is a review of relevant military and artificial intelligence (AI) literature and systems. Chapter III presents the Conceptual Planning Framework, which includes several concepts and definitions used in the following chapters. Chapter IV describes the fire support planner as a conceptual model using mathematical notation and pseudocode, without designating a particular computer programming environment. It includes an asymptotic analysis of the model's running time. Chapter V presents a prototype implementation of the fire support planner in a selected combat simulation environment and programming language, and Chapter VI describes the testing process and results for the prototype. We offer conclusions and ideas for further research and development in Chapter VII. Appendix A describes the combat model we developed as a platform for testing our contributions, and Appendix B provides additional details about the fire support planner implementation. Appendix C provides a review of doctrinal military

planning and operational concepts. Appendix D includes the details of the testing scenarios mentioned in Chapter VI.

C. BACKGROUND

This project is at the intersection of three separate disciplines, each with its own lexicon: modeling and simulation (M&S), artificial intelligence (AI), and military planning. To avoid confusion between overlapping terms, we present some definitions here that apply throughout the document. All quoted text in this section is meant to be definitional.

1. Combat Modeling and Simulation

M&S is “the discipline that comprises the development and/or use of models and simulations” (DOD 2013a). Generically, a *model* is “a physical, mathematical, or otherwise logical representation of a system, entity, phenomenon, or process” (DOD 2013a). This work is focused on *combat models*, which are models of military (or paramilitary) forces using maneuver and violence to achieve goals. Combat models and the simulation processes that employ them can be used to support a variety of activities, such as acquisition, experimentation, operational decision support, and training, but our work is not specific to any of these purposes. Where necessary, we use the term *analysis* as a collective term for all uses of combat models and simulations except training—which tends to have its own unique considerations. When we say *analysis and training*, we mean all uses of combat M&S.

We prefer the DOD definition of *simulation*, “a method for implementing a model over time” (DOD 2013a). In practice, however, the word takes on various meanings—for example, we have seen it used to mean both a combat model implemented on a computer and a single execution of such an implementation. To prevent confusion, we avoid using the term *simulation* in isolation.

Here are the relevant vocabulary terms for combat M&S-related concepts. We define more specific version of these terms shortly.

- *Simuland*: “The system being [modeled]” (Gross 1999, 67)

- *Referent*: “A codified body of knowledge about a thing being [modeled]” (Gross 1999, 65)
- *Conceptual Model*: “A collection of assumptions, algorithms, relationships, and data that describe a developer’s [solution]” (DOD 2013a). Although M&S management literature often describes a conceptual model as an informal description of problem-solving requirements, it suits our purposes to define it as a collection of designs and algorithms written in formal mathematical language. Conceptual models are not meant to be executable.
- *Executable Model*: “A model that can be executed, [usually] a computer program. Execution of the executable model is intended to simulate the simuland as detailed in the conceptual model, so the conceptual model is thereby a design specification for the executable model, and the executable model is an executable implementation of the conceptual model” (Petty 2010, 328).
- *Simulation Time*: Time measured in the world of the model, as opposed to real-time
- *Entity*: “A distinguishable person, place, [or] thing...about which information is kept” (Gross 1999, 52). In combat models, an entity is an individually modeled soldier, vehicle, animal, autonomous piece of equipment, or group thereof (but see *unit*, below). Although it may have logical or geometric components, an entity is treated as a single, indivisible object.
- *Unit*: For combat models, a unit is a representation of one or more people and their equipment that work together on a shared task in close geographic proximity. A unit may contain entities, but it may not contain other units (the last point is a departure from the way military professionals define units, but see *command* below).
- *Aggregation*: In some combat models, units are modeled as atomic, indivisible objects—their contained people and equipment are abstracted away—in which case *unit* and *entity* mean the same thing. This kind of abstraction is called *aggregation*. When units and entities are equivalent, we refer to them as units.
- *Singleton unit*: If the combat model has entities that do not group together into units—that is, individual entities that act alone—then we can refer to the individual entities as *singleton* units. In most cases, we can drop the word *singleton* and just call these entities units. The idea here is that we still conduct our reasoning at the unit level when a unit is comprised of just one entity.

- *Command*: A collection of units with hierarchical command relationships. Our definitions of *unit* and *command* differ from typical military usage, which treats them as synonymous except that the latter term is only for battalion-sized units and larger. We use *command* even for groups of small units, such as a platoon—i.e. a platoon headquarters in command of a few squads. The entire platoon is a command, but the platoon headquarters (without the subordinate squads) is a unit. This definition of a command helps with clarity of language for modeling.
- *Ground-based*: affixed to a terrain skin, but may move across it. By definition, all entities in a ground-based unit are ground-based entities. A characteristic feature of ground-based units is limitation of visibility and movement by terrain.
- *Scenario*: an input configuration for an executable model. For combat models, this includes data such as the terrain map, entity configurations (weapons, reactive behaviors, etc.), unit locations, and preplanned actions. A scenario may model a real world or fictional situation.
- *Replication*: synonymous with *simulation run*, a single execution of an executable model configured with a particular scenario. We use the term for both real-time (i.e., most training models) and non-real-time (i.e., analysis) execution modes.

2. Battle Planning

Our ultimate objective is to produce realistic plans executable by units in a combat model implemented on a computer. Therefore, we take an AI point of view to define *planning* as the formulation of a partially ordered set of instructions for a set of agents in an attempt to modify the world state from an initial configuration towards a desired outcome. This is a more general definition than that of classical state-space AI planning because we will need extensions such as plan-space, hierarchy, and continuous time (we explain how the combat modeling domain violates several classical planning assumptions below).

Battle planning is a type of planning where the agents are units in a combat model, the initial configuration is a scenario, and the desired outcomes are *military objectives*. The latter may include *goals*, given as a logical combination of predicates evaluated against the world state, as well as numerical *objective functions*, which should be optimized. An *automated battle planning system* (ABPS) is a software system (or

subsystem) that accepts a scenario and some military objectives as input and produces a battle plan that is executable by designated units. This work is focused on battle planning for ground-based units. Due to the unpredictability of most combat models, the designated units are not guaranteed to achieve all military objectives by executing the plan—but the *effectiveness* of the ABPS is the degree to which its plans are successful at doing so. Effectiveness is just one part of an ABPS’s *validity*, which is discussed below.

It is convenient to distinguish an ABPS, assuming we have one, from the rest of the executable model. To that end, we define a *combat simulation environment* (CSE) as an executable combat model, along with its supporting tools (for scenario design, behavior development, etc.), with the exclusion of any ABPSs.

We assume that an ABPS is only useful if it produces battle plans that resemble those of human planners. In other words, an ABPS is a model of a human commander and, if appropriate, a planning staff. Therefore, the development of an ABPS is an M&S activity subject to M&S control and evaluation processes. Since this work is restricted to ABPS development, we assume that the CSE is a fixed version (i.e., the development team is not modifying the CSE at the same time the ABPS is being created or changed). This lets us narrow our focus for issues such as verification and validation to the ABPS. More specific definitions related to ABPS development follow.

- *Planning Simuland*: This is what automated battle planners represent—real world military and paramilitary commanders and planning staffs of tactical units, in the context of their development of battle plans. In particular, we are interested in the fire support planning concepts and processes of these groups.
- *Planning Referent*: This is what we use as the authoritative source to guide automated battle planner development—doctrinal, training, and technical publications of U.S. and other nations’ military organizations. In particular, we focus on the fire support planning concepts, which are found in a variety of different documents (not exclusively in fire support publications). An overview of the referent’s literature is provided in Chapter II.
- *Conceptual Planner Model*: A collection of data structures, logical components, and algorithms used to model the plan-generating capability of commanders and planning staffs, written in mathematical exposition, software engineering diagrams (such as UML), and pseudocode.

- *Executable Planner Model* or *ABPS*: An executable implementation of the conceptual planner model for a particular CSE.
- *Unit* and *command*: Units and commands are defined for an ABPS the same as they are for a combat model. However, the ABPS need not be provided ground-truth unit information contained in the CSE.
- *Scenario*: Scenarios have the same meaning in an ABPS context as they do for combat models in general. The ABPS may be provided an input scenario different from the current ground-truth configuration of the CSE. It generates plans using whatever scenario data is provided to it as input.
- *Invocation*: For the ABPS, we replace the term *replication* with *invocation* to avoid confusion. An invocation of the ABPS is a single execution of its planning capability, resulting in an executable battle plan (or failure).

Initial planning is the generation of the first plan in a replication for a set of units. Initial planning may be done offline (prior to a replication) or online (during a replication, prior to the units' first planned actions). An offline-generated initial plan could be reused for many replications of a stochastic combat model (using different random seeds) or for different human training audiences. *Replanning* is planning activity for units that already have a plan, including both modification and complete replacement of the plan. In theory, an ABPS could replan several times during a single replication. Depending on the design of the combat model, there could be more than one ABPS for a scenario (one for each of the opposing forces, for example, or even one per unit). These variations demonstrate that the number of invocations of an ABPS is usually not equal to the number of replications of a particular scenario.

3. Human Behavior in Combat Simulations

The aspects of a combat model can be categorized as either physical or behavioral. Most of the physical processes involved in combat are well understood, so the physical aspects of our models can essentially be made as accurate as desired. Usually, the only limitation is the computational budget. Human behavior models, on the other hand, are generally poor representations of human battlefield behavior. Because of this, scenario designers and exercise managers tend to either carefully control the modeled human behavior (with many human controllers) or carefully validate scripted behaviors for one-time use.

Automated battle planning is a subset of behavioral modeling distinguished by forward reasoning, which means that planners must be able to predict the results of actions in order to choose a series of actions that will lead to a desired outcome. In contrast, *reactive* behavior models only select a single response action or state based on the current state, with very short-term or no reasoning about future effects. The potential value of planning over reactive behavior is that a series of actions with lower short-term utility may produce better results than a series of reactive decisions with optimal short-term utility. Furthermore, human commanders and planning staffs *do* reason about the future, and they *do* formulate and carry out battle plans to orchestrate the employment of units. A reasonably accurate model of military behavior above the small unit level, therefore, must model planning. Without an automated way to do this, human designers or operators must “manually” simulate the planning process for the modeled units.

The potential drawback of planning, especially with stochastic combat models, is that outcomes may diverge from the planner’s predictions. Human commanders know this; in the often quoted words of Helmuth von Moltke the Elder, “no plan survives contact with the enemy” (Barnett 1963, 35). A plan whose predictions turn out to be incorrect during a replication could easily result in worse performance than purely reactive behavior. An important point, though, is that plans generated by humans—by scenario designers, for example—are just as subject to these variations.

4. Current Processes for M&S Battle Planning

The prevailing solution for generating battle plans in combat simulations is to use meticulous human input. We describe the generic human processes for non-real-time and then for real-time applications.

a. Battle Planning for Non-Real-Time M&S

For non-real-time M&S, a small team of human designers receives a request for a study or training artifact. First, the team (usually together with the requesting agency and other information-providers) invents a set of abstract scenarios, each differentiated by a type of terrain, composition of forces, mission objective, etc. The next step is to gather relevant data such as terrain maps, weapon effectiveness measures, and tactical doctrine.

The team uses this data to build scenarios, first placing units in “bins” without plans. Once the scenarios are fully constructed (with the exception of plans), the designers consult with tactical experts external to the design team, who help them develop a *scheme of maneuver*, or basic concept for a battle plan. The designers encode the scheme of maneuver into a battle plan in the language of the executable model. The design team then iteratively refines the plans by running the executable model and observing its output, attempting to correct unreasonable behavior by the modeled units. Once the design team is satisfied, the final plans may be put through another validation by the same, or other, tactical experts. Finally, the scenarios (with plans) are declared ready for data collection or training events.

By nature, this process does not support replanning. Designers may build a small number of conditional branches into their plans if they notice discrete possible outcomes at key points, but each such branch requires validation. Battle planning is not the only time-consuming part of the process, but it often comprises a significant portion of the scenario designers’ efforts. Other time-consuming steps include determining the scenario requirements from the sponsor, determination of opposing force composition by external agencies, collection and verification of input performance data, verification and validation, and development of new reactive behaviors.

b. Battle Planning for Real-Time M&S

Most real-time M&S falls within the training domain. The initial planning for a real-time event may follow the same process as the initial planning for non-real-time M&S, but there is typically less time available for quality control. In many cases, the designers use a validated (or, less formally, “time-tested”) scenario as a starting template, then make adjustments based on the event requirements.

Once a real-time replication begins, changes to plans become necessary. As events play out contrary to planning assumptions, the training audience is likely to change its own plans. Other modeled units, not controlled by the training audience, need to change their plans for the same reason. Units must also react to the changed plans of higher, adjacent, and opposing units, resulting in a complex feedback loop.

For the units not controlled by the training audience, which can include friendly, enemy, and neutral elements, these changes are handled by numerous human operators. These individuals must be well trained on the particular combat model, and they must have tactical acumen to simulate realistic commander decisions. The number of operators needed depends on the level of granularity required for the event. Usually, the operators handle both low-level reactive behavior (which is not our primary focus) as well as replanning for larger units as the scenario plays out. In many cases, a single person plays the role of several different unit commanders, switching contexts as required by events. An alternative to changing plans is to constrain events such that initial plans (perhaps with some branches) do not become invalidated. This may be done by disallowing unexpected decisions in the input stream (i.e., the training audience) or by an empowered operator arbitrarily modifying the execution state—changing unit locations, resurrecting entities, and so on.

5. Fire Support Planning

The technical contribution of this work is an automated fire support planner. Before defining *fire support planning*, we need to introduce some related terms. According to U.S. military doctrine, planners must consider six *warfighting functions*: maneuver, fires, command and control, military intelligence, sustainment, and protection (DOD 2008, 2010, 2015). We focus on two of these functions. *Maneuver* is “employment of forces in the operational area through movement in combination with fires to achieve a position of advantage in respect to the enemy” (DOD 2015b, 145). *Fires* means “the use of weapon systems or other actions to create specific lethal or nonlethal effects on a target” (DOD 2015b, 86). *Fire support* is slightly more specific: “fires that directly support...[friendly] forces to engage enemy forces, combat formations, and facilities in pursuit of tactical and operational objectives” (DOD 2016b, 86).

Direct fire uses “the target itself as a point of aim” (DOD 2016b, 68), while *indirect fire* is directed against targets that the weapon operator generally cannot see, using some other unit’s target location data. Direct fire weapons include rifles, machine guns, tank main gun, and shoulder-fired rockets; indirect fire weapons include mortars,

howitzers, and rocket artillery. Some types of weapons, such as naval ship guns and precision-guided bombs, can provide either direct or indirect fires, depending on how they are employed. In this work, all direct and indirect fires are considered fire support when directly contributing to another unit's objective. Note that although some publications list only indirect fire and close air support under the category of fire support (DOD 2006, 10–3), the concept of direct fire support can be found in both Marine Corps and Army doctrine—for example, *Marine Corps Warfighting Publication* (MCWP) 3–11.1 (DOD 2014a, 1–10) and *Army Techniques Publication* (ATP) 3–21.8 (DOD 2016a, 2–58).

Fire support planning is conducted at all echelons of command. At the small unit level, a squad leader may place a fire team in a support by fire position while the rest of the squad assaults an objective. In much larger scale, a joint task force may plan a weeklong air campaign to degrade enemy ground forces prior to a large-scale attack. Our work focuses on smaller echelons and time scales, but we use echelon-generic terms where possible.

Following Hughes (1995), we differentiate between two possible effects of fires: *attrition* and *suppression*. Both describe a reduction in the effective combat power of the units fired upon; attrition is a permanent effect while suppression is temporary. Attrition is easily represented in the typical combat model by the elimination of entities (and their ability to fire) from units or the reduction of a unit strength attribute. Suppression is explicitly defined in doctrine—“temporary or transient degradation by an opposing force of the performance of a weapons system below the level needed to fulfill its mission objectives” (DOD 2016b, 229)—and is always considered by real world tacticians as a primary conceptual planning tool. Anyone that has been the target of live fire understands the effectiveness of suppression—there is a certain amount of attention to one's own survival that gets in the way of returning fire effectively. Furthermore, dust clouds, noise, and overpressure from nearby impacts make the targeting task more physically challenging. Despite its universality in real world combat, suppression is often ignored in combat models, especially those used for analysis. Modeled units tend to retain their

same capabilities when fired on, with combat power affected only by attrition. For some examples of suppression models, see Chapter II, Section E.

The canonical example of fire support, at the scale of interest here, is the *support by fire* task: “[engaging] the enemy by direct fire to support a maneuvering force” (DOD 2014a, D-2). This concept is illustrated in Figure 1, where we see a unit of three armed vehicles (behind the red icon) suppressing a defending unit from a base of fire while the remainder of the attacking force moves to assault the objective. A domain expert would expect to see this fundamental infantry tactic in a model of combat; unfortunately, it is not very common. Note that this example shows direct fire weapons employed to suppress the threat, but indirect fire weapons could achieve similar effects if given accurate target location.

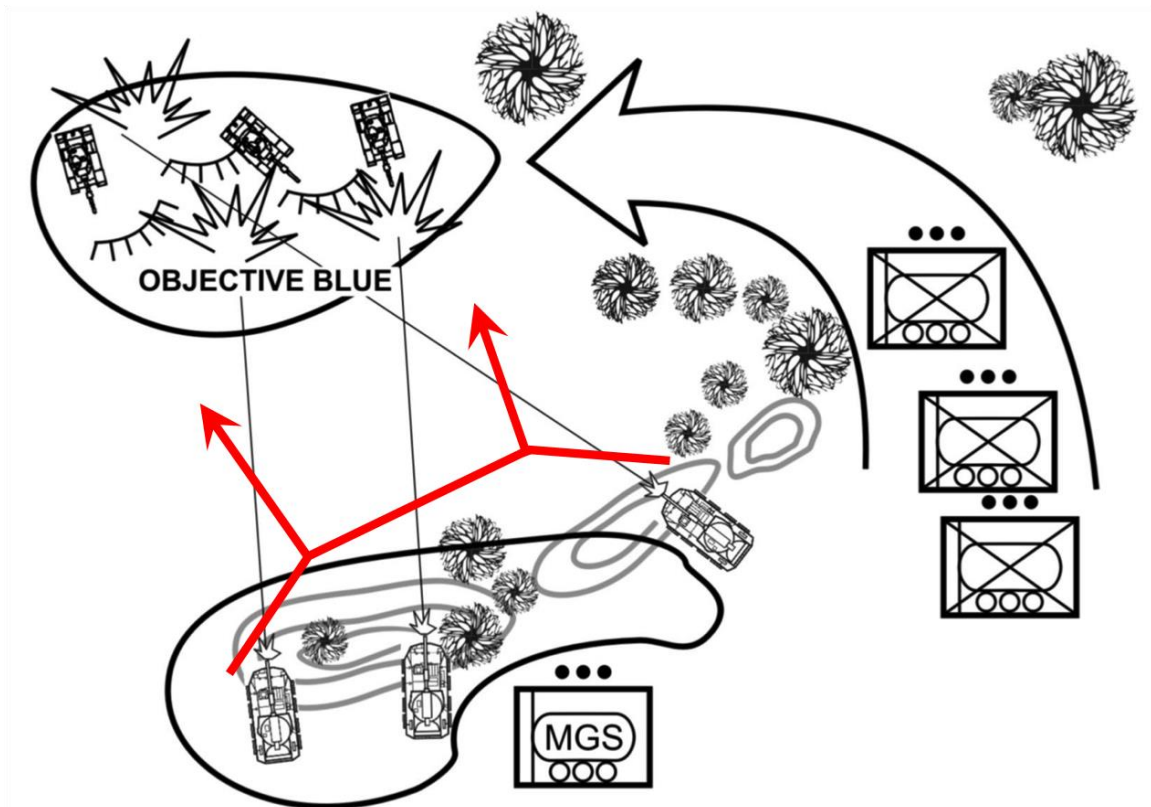


Figure 1. Suppressive Fires in Support of an Assault. Adapted from FM 3–21.11 (DOD 2003).

D. PROBLEM DISCUSSION

Today's executable combat models are not capable of automatically generating realistic battle plans. Human scenario designers and operators perform the task instead. For M&S-based analysis, a significant portion of the time required to complete a study is taken by the battle planning effort. This limits the number of scenarios for a study to only a few. For M&S-based training, the lack of automated planning capability increases the requirement for human operators, limiting the efficiency gains that seem to be promised by computer-based alternatives to live training. Although a few examples of automated maneuver planning have appeared in military and video game research, there is only one example of automated fire support planning with a focus on suppression (Shi and Crawfis 2014), and its simplifying assumptions make it inapplicable to the type of combat models in use today.¹

1. Limitations to Analysis

Robust design is the idea that a “system must be relatively insensitive to uncontrollable sources of variation present in [its] environment” (Sanchez 2000, 69). The reference here is to a real world system, which no doubt will be employed in a variety of different environmental and tactical situations over its operational life cycle. Prior to development or procurement, requirements analysts and systems engineers use M&S to evaluate system alternatives. The alternatives being modeled might perform particularly well or badly in the handful of chosen scenarios, but differently in other scenarios not considered, leading to biased analysis that may not be detected until the system is employed in real world conditions.

Although battle planning is not the only time demand in the analysis process, it is one of the only steps that must be redone for every scenario. Combat resolution data (i.e., probability of hit), for example, can be reused across many scenarios and many battle plans, with a one-time collection cost. Improving the automation of battle planning would provide a significant efficiency gain for analysis. It could either reduce the time to prepare studies or broaden their apertures. Even a single scenario, with fixed terrain,

¹ See section F.4 of Chapter II.

force pairings, and starting unit locations, could provide a more robust analysis if alternative battle plans are explored.

2. Automated Replanning

The human-powered battle planning process does not efficiently support replanning during a replication. Without automated replanning, designers have the following options for dealing with plans whose predictions do not hold:

- (1) Use carefully restricted scenarios in which replanning is unlikely to be needed, limiting the robustness of the study or the value of the training event.
- (2) Accept that modeled units will continue to follow a battle plan that has been overcome by events, even though no human commanders would ever do so. This calls into question the validity of the study or the believability of the training aids.
- (3) Detect when a plan has become useless, and then use human input to replan. This approach is only used for real-time training applications, where option (2) is unacceptable due to the training audience's observation.

For analytical studies, options 1 and 2 may produce reasonable-looking but biased numerical output, leading to biased recommendations. Option 3 is undesirable due to its cost in man-hours.

3. Training Implications

For training, exercise controllers must trade off the size of the modeled forces or the level of realism they exhibit according to the number of human operators available. As a result, military sensing, reconnaissance, and targeting capabilities have progressed beyond our ability to train on them. For example, in a conventional warfare role, intelligence analysts are expected to infer large enemy unit formations and types based on observations of small parts of those units moving in relation to each other, but we cannot generate realistic and novel small unit behaviors on-demand without many human controllers. In a stability operations role, determining the purpose of small unit actions may be of greater interest, but acting with purpose usually means following a plan, at

least a short-term plan developed informally. If we require as many human operators as modeled decision-makers, then M&S systems are not living up to their resource-savings promise. We may have even lost fidelity, since a live camera and an appropriately dressed role-player would do better than a virtual environment in replicating the real world.

In many M&S training environments, the options for the training audience are limited to what pre-existing plans can support. This is a limitation of live training, too, since training areas and safety regulations only go so far. Whether constraining the decisions of a training audience is appropriate depends on the training goals. For example, if a training audience has specific situations that must be presented for a scripted evaluation, then a forced sequence of events is acceptable. But if tactical insight and creativity are the goals, then M&S should provide *more* flexibility than live training. Without automation, this flexibility requires large numbers of operators.

4. The Foundational Role of Initial Planning

Both initial planning and replanning are necessary to resolve these issues fully. However, much of the technology for initial planning is also likely to be useful in replanning. The trust of the modeling community in the effectiveness and realism of these envisioned tools is essential for their adoption and continued development; therefore, comparison of automated initial planning to human initial planning is a necessary step. Automated initial planning is the more foundational capability, so it is the focus of this work.

5. Formalizing the Planning Problem

We now describe automated battle planning in the terms of automated planning in the AI field. This helps guide us toward useful technologies and identify open problems. Classical state-based planning is the academic entry point to this domain, so we begin by considering the eight assumptions that define classical planning problems (Ghallab, Nau, and Traverso 2004, 9–10). To do this, we must make some assumptions about the combat models of interest. We remain as generic as possible to maintain the general applicability of this work. We present the eight assumptions as questions; the answer to each of these would be “yes” for classical planning.

- Finite state space? No. For all practical purposes, the state space of a combat model is infinite. It includes unit or entity locations in two- or three-dimensional space, and the time model is either subject to continuous subdivision (as in a discrete event model) or has such a small discrete subdivision that we cannot hope to examine every possible state of the model or every possible partially ordered set of actions.
- Fully observable state? Assumed. In general, the ground-truth data in the entire CSE is not provided to a planning agent. This models the limitations of military intelligence in the real world. However, to focus on certain aspects of the problem, we do provide the planner with ground-truth unit locations and combat resolution parameters such as probability of hit.
- Deterministic action effects? Assumed. Many combat models are stochastic, which immediately discards the notion that we can predict the CSE state as the result of an action with certainty. However, in some combat models only some types of outcomes, such as the effect of firing a weapon, are random. Our ability to predict the approximate system state forward in time depends on the number of expected stochastic outcomes up to that time. Our general approach, discussed further in Chapters III and IV, is to reason with an abstract, deterministic model of the combat model—a second derivative of reality, in some sense—using probabilistic concepts such as most likely outcomes and expected values. This approximation cannot be assumed sufficient without testing in the executable model. If we are planning for a completely deterministic combat model, we still may need more abstracted, computationally efficient models for planning, but either way we are working with deterministic action effects in the conceptual world of the planner.
- Static system? Assumed. Since each ABPS plans for only one of the competing sides, we cannot assume that the actions in the plan are the only events that can change the CSE state. For this project, we assume that enemy units do not move. Although stochastic effects are unpredictable, we assume that the reactive tactical decisions (whether to fire, which targeting mode to use, etc.) of units are predictable enough that we can estimate the CSE state some reasonable amount of time in the future with non-vanishing probability.
- Simple goals? No. The desired outcomes of a combat model can almost never be described simply as a combination of predicates describing the end state. Users insist that units follow reasonable tactics, make decisions and exhibit behaviors that resemble those of their human counterparts, and often need utility functions to describe required processes or resource consumption.

- Sequential planning? No. Units in a combat model act simultaneously, so a total order on actions, as in classical planning, is not a correct or useful model of a battle plan.
- Time represented only by states? No. Many combat actions have durations measured with explicit time, which is awkward or impossible to represent with a purely state-based measure of time. If resources such as fuel or ammunition are involved whose consumption depends on nontrivial functions such as the force due to gravity, then state durations are not a useful measure of time. Even in a fixed-time-step combat model, the size of the time step is often too small to be used as the duration of a planning state. The behavioral state of each unit—for example, its current movement path, tactical formation, and targeting mode—changes at a non-steady rate depending on features such as distance and geography, further limiting the usefulness of a state representation of time.
- Offline planning? Assumed. We addressed above the importance of online replanning. Offline planning here means that the CSE state cannot change *during* an invocation. Although a real-time application may need to deal with this possibility, we assume in this work that we can freeze the executable model or perform planning prior to a replication. We maintain the importance of online replanning, but we focus on offline initial planning as the foundational capability.

With most of these assumptions violated, it is clear that pure classical planning methods are not appropriate. In particular, state-based planning does not appear useful, a conclusion similarly argued by van der Sterren (2013). We discuss alternative planning techniques in Chapter II.

E. CONTRIBUTIONS

Battle planning requires a variety of different types of reasoning, as illustrated above by the six warfighting functions. Even within a single function, a variety of data structures and algorithms seem useful for different situations. Our approach is to assume that an effective ABPS must bring together a variety of functional components, not settle on one universal approach. Our Conceptual Planning Framework is a foundational step in architecting a multi-component planner. Since some effort and success has been put towards the ground maneuver planning problem (as discussed in Chapter II), and little has been done to automate fire support planning, we focus on the latter for the detailed part of the work.

We describe a natural way to model a fire support planning problem and an algorithm that handles it well in polynomial time. The effectiveness of the fire support planning component supports the argument that reasonable automated battle planning capability is achievable. Comparison of the fire support planner to existing maneuver planners indicates that different planning functions are best automated with different approaches, supporting the argument that a federated approach to battle planning is both appropriate and practical.

Our fire support planner itself could stand alone as a useful tool in some applications, either offering fire support plan options to users or providing opposing fire support plans for analysis. With an implementation effort guided by the prototyping done here, the fire support planner could be incorporated into existing combat models with similar structure. Alternatively, some of its novel data structures and algorithmic approaches may be useful in related work.

The specific, novel contributions of this dissertation are

- The *Conceptual Planning Framework*, a design scheme for integrating, managing, and employing multiple automated planning components in support of complex combat models
- A *generalizable mathematical model of tactical risk and fire support* including the risk object hierarchy, availability tasks, fire support tasks, and affected risk sets, along with well-defined manipulations for the reduction of risk and other planning-related effects
- The *fire support planning algorithm*, a polynomial-time greedy best-first plan-space planner with a simple but effective heuristic for keeping fire support plans simple and realistic, which operates on the mathematical model
- The *Wombat XXI* executable combat model, a relatively simple but relevant research platform built on the Unity 3D game engine, that resembles several tactical combat models currently in use. Compared to production combat models, it is more accessible and easier to use.
- A *prototype ABPS implementation*, conformant to the Conceptual Planning Framework, as a research platform for architectural ABPS design and development of new battle planning components

- An *implementation of the fire support planning algorithm* within the prototype ABPS, demonstrating how the theory can be effectively applied to a realistic and stochastic execution domain
- An *iterative testing and development process* that can be used to support verification and validation of ABPS components
- A *quantitative and qualitative evaluation* of the fire support planner implementation, demonstrating its effectiveness against simple but reasonable plans manually constructed by the author

We discuss some extensions to the fire support planner in Chapter IV and broader possibilities for future work in Chapter VII.

F. NEED FOR STUDY

Table 1 lists recent research requests from various organizations. These demonstrate a continued interest in improving behavior modeling for combat simulations.

Table 1. Documented Research Requirements.*

Item	Code/Category	Title	Description	Research Question
Source: <i>Naval Research Program</i> (https://my.nps.edu/web/naval-research-program/)				
1	NPS-17-M216	Integrating Engineering Models in Analytical Constructive Simulations	Determine the feasibility of integrating acquisition community models into analytical constructive simulations to assess likelihood of programmatic success and conduct of operational requirement tradeoffs. (Common to both proposals.)	What updates to behavior models are necessary to accurately depict order of battle, concept of employment, and concept of operations?
2	NPS-17-M173	Using Simulations to Improve Combat System Design		How does current scientific and engineering literature on behavior modeling products apply to systems engineering and analytical processes?
3	NPS-N16-M159	Commercial Gaming Industry Best Practices	Examine the gaming industry for innovations below the individual application level that are applicable to the DOD modeling and simulation community.	What technology advances does the commercial game industry implement that can be leveraged by USMC modeling and simulation programs of record?

Item	Code/Category	Title	Description	Research Question
4	NPS-N16-M163	Exploring Potential Alternatives Using Simulation and Evolutionary Algorithms	Develop methods and software tools that enable broad-scale search for alternatives/strategies in a simulation scenario	[Iteratively] create the next set of solution vectors to try, stopping at some user-specified criteria.
5	NPS-N16-M199	Developing analytic capability in Combat XXI	Embed Combat XXI in a data farming environment to enable efficiently running and analyzing large-scale designed experiments. This will allow USMC analysts and decision makers to get better and timely insights from their simulation models.	Develop methods, designs, and software tools that enable efficient investigation and post-run analysis with Combat XXI.
6	NPS-N16-N198	Enhancing STORM Analytic Utility	Develop methods and software tools that enable efficient investigation and post-run analysis with STORM.	Facilitate analysts' ability to quickly glean insights about the critical relationships within a campaign.
Source: 2012 U.S. Marine Corps Science & Technology (S&T) Strategic Plan (https://www.onr.navy.mil/~media/Files/About-ONR/USMC-ST-Strat-Plan-2012-Final-31Jan.ashx)				
7	T&E STO-5	Adaptive Simulated Entities	Simulation capabilities are needed to enable Marines to train as they fight, using the full range of equipment functions in training.	Develop technologies that address the Marines' need to train kinetic and nonkinetic skills in complex, simulated environments containing realistic, adaptive entities. Develop behaviorally accurate virtual role players representing a variety of cultures, populations, and domains that would allow Marines to apply learned skills.
Source: U.S. Marine Corps OST&E Unified Priority List (UPL) (available by request from Headquarters, Marine Corps)				
8	2016 USMC OST&E UPL	Includes related capability gaps not printed here due to FOUO marking.		
Source: Office of Naval Research (ONR) (http://www.onr.navy.mil/About-ONR/science-technology-strategic-plan.aspx)				
9	Naval S&T Strategy	Appendix A Focus Area: Autonomy & Unmanned Systems	Objective: Intelligence Enablers and Architectures	Scalable planning and re-planning

Item	Code/Category	Title	Description	Research Question
10		Appendix A Focus Area: Warfighter Performance	Objective: Bio-engineered Systems	Design computational cognitive models for intelligent systems and synthetic forces for operational experimentation, mission planning, real-time decision support and training systems
11			Objective: Human Systems Design and Decision Support	Develop effective, user-friendly decision support systems for kinetic and non-kinetic operations
Source: <i>ONR Code 30: Expeditionary Maneuver Warfare & Combating Terrorism</i> (http://www.onr.navy.mil/Science-Technology/Departments/Code-30.aspx)				
12	ONR Code 30, Maneuver: Willful Intent	Autonomy	In stride support of USMC rifle squads to include tactical decision making while in enemy contact Set-it and forget-it enhanced mission planning tools	Developing algorithms that produce a wider range of robust and tactically appropriate behaviors
13	ONR Code 30, Human Performance Training and Education: Willful Intent	Decision Making & Expertise Development	Prototype simulations to support training adaptability and decision making.	Lack human performance assessment technologies that reduce instructor/ student workload & contractor support.

*Information adapted from the web-based sources listed in table; all text is quoted directly.

Item 1 specifically calls for behavior model upgrades for combat units. Since battle planning and fire support planning are critical functions of all command echelons, this work is relevant to that goal. Item 2 is a sibling research request focused on systems engineering; we argued above for the potential benefits of automated planning in a model-based systems engineering context.

Item 3 is a request to apply relevant video game technology to military M&S. Although automated planning has not reached a level of ubiquity in industry products, it has been employed in a few well-documented titles (see Chapter II) and remains a subject of interest in game AI research circles. Real-time strategy (RTS) games, and their “game-playing bot” competitions, continue to receive a central focus in that community. RTS games resemble combat models in several important ways, and they have been the

subject of several planning approaches. We discuss RTS-related research further in Chapter II.

Although item 4 seeks an evolutionary algorithm-type solution, we note that the deeper goal is to explore an operational design space. The request seems to presuppose an automated planning capability inherent in the evolutionary approach. Although evolutionary algorithms have been employed for this purpose (we discuss examples in the next chapter), they are just one of several different tools that could be employed for automated planning. In Chapter III, we offer an architectural framework that could include evolutionary algorithms along with other planning techniques.

Items 5 and 6 are after improved analytical payoff from specific combat models. Automated planning is not specifically mentioned in either, but it is an additional tool that could expose new dimensions in a study's problem space in an efficient way. Both requests emphasize rapid results, which agrees with our objective of efficiency through planning automation.

In item 7, the U.S. Marine Corps is interested in improving its human behavior models for training applications. This is a broad capability requirement, spanning the range of military operations, including combat operations of varying intensity. In particular, for the conventional warfare aspects of training, effective automated battle planning would be useful in representing both opposing forces and friendly forces where human role-players are not available. Item 8 lists similar capability gaps.

The Naval S&T Strategy covers ground, sea, and amphibious capability gaps somewhat generically. Items 9 and 10 specifically call for improvements to planning systems, including the vision of a future force with a mix of human and robotic capabilities. Automated battle planning, at least at some echelons of control, is clearly relevant for the design of operations with automated combat systems. Item 11 refers to "decision support," which indicates the possibility of using M&S to aid in real world operational planning (as opposed to fictitious analytical or training scenarios). In this role, the timesaving benefits of automation are even more critical.

Items 12 and 13 are from ONR Code 30, which primarily supports U.S. Marine Corps research requirements. Item 12 further indicates a need for real world planning support with the inclusion of automated combat systems. Item 13 has a training focus, with two important gaps. First, the need to train thought processes and improve the quality of leaders' problem solving calls for a more open and reactive environment—which implies either a large number of role-players or a replanning capability. Second, the reduction of manpower without losing tactical quality can only be achieved with better automation.

G. CHALLENGES AND POTENTIAL IMPACT

A potential counter-argument to the use of automated planning and replanning—that is, inertia of the human-centric status quo—is that automated planners are not capable of replicating the kinds of tactical decision-making that human commanders or scenario designers can. The results we achieve for automated fire support planning, described in Chapter VI, argue against this claim. Furthermore, observe that modeled units failing to adapt their plans to the situation are no less behaviorally faulty. This is a common issue for static plans in combat simulations. Donaldson provides some illuminating examples (2014, 105) at the entity level; our concern is with broken plans at the larger unit level.

For the next point, assume an analysis context in which we are exploring several future system alternatives. If we assume the existence of an effective ABPS capable of online replanning, then we may have another problem: the adaptability of the model could reduce the statistical variability between the system alternatives. The modeled units would adjust plans for the capabilities present in each scenario, perhaps leading to similar results at all design points. This could make the analyst's task of finding statistically supported recommendations more difficult. In a sense, this is the dual of the uncertainty-modeling problem described by Hughes: "By including uncertainty about the human element we hope to [move] the center of gravity of results toward 'truth,' but...at best we will have a more accurate mean but a wider variance" (1994, 50). His statement refers to a single design point. If we are comparing several design points in a high-uncertainty

environment, the concern would be that the true *difference* in means could become undetectable by a hypothesis test (due to variance remaining in the error term). In contrast, effective replanning should (still) improve the accuracy of the mean and *reduce* the overall random variance per design point, but it would potentially reduce the difference in means *between* design points due to unit adaptability. The result for both problems, in theory, is a failure to draw statistically significant conclusions about means.²

If so, then what does the simulation output have to say about the system alternatives? M&S analysts are not useful to decision-makers if their answer for every study is “the statistical evidence does not support the hypothesis that the mean for system A is different than the mean for system B at a significance level of α ” (although some alternatives really do exhibit comparable performance under realistic operating conditions, a possibility that should not be discounted). To get past this, they must be able to broaden the aperture of their analysis to describe the *range of conditions* under which certain alternatives seem to be better or worse, and which experimental variables are the most significant in causing variation. This is the kind of insight that models are supposed to provide, not simple “A or B” answers (Box and Draper 1987, 424). The true value of analytical combat models requires space-filling designs of experiments in which the scenario itself is a factor, which is intractable with a human-centric battle planning process. The only hope for achieving this level of analysis is a valid automated battle planning capability.

² Difference in means is not the only statistic of interest, but it is the one used most often.

THIS PAGE INTENTIONALLY LEFT BLANK

II. LITERATURE AND TECHNOLOGY REVIEW

A. OVERVIEW

In this chapter, we review some of the important literature and demonstrated systems related to automated battle planning. We begin by describing some basic tactics for ground operations to provide a framework for the modeling and planning work in the next two chapters. We then focus on the AI field, reviewing foundational automated planning theory. Next, we cover some additional AI-based behavior modeling techniques, outside the normal scope of the planning domain, that are relevant to the work ahead. We move on to a high-level description of several production systems to help frame the broad class of CSEs where the present work is most directly applicable. We conclude this chapter with a review of some published research projects and industry-leading products that actually employ automated battle planning.

B. TACTICS

We limit our consideration in this work to infantry units. To ensure a common understanding of how an infantry attack is conducted, we summarize some basic tactical concepts in this section. Additional, related military processes and techniques are reviewed in Appendix C.

1. Tactical Considerations for an Infantry Attack

Infantry units take advantage of terrain more than mounted units do. Dismounted troops are slower than vehicles, but can cross nearly any kind of ground given enough time. This often provides a wider range of attack axes than armored, mechanized, or motorized units can achieve. However, infantry are more susceptible to high rate of fire weapons and area fired weapons, such as machine guns and artillery. Without the protection of vehicle armor, infantry seek to optimize the use of cover and concealment. When that is not possible, infantry commanders attempt to neutralize or suppress threats using fire support, mutually supporting maneuver units, or both.

The most decisive step in an infantry attack is usually the assault, during which designated assault units literally move into and through the enemy positions, eliminating opposing forces through fire and close combat. Most aspects of an infantry attack plan are in support of getting the assault units onto the objective.

Where infantry may be exposed to enemy fields of fire, the commander plans to eliminate or suppress the threat. If there are more targets than fire support assets, the commander will look for a sequence of fire support missions in coordination with the scheme of maneuver. Such a sequence must account for asset displacement, target shift time, and—in the case of aircraft—time on station limitations (due to fuel consumption). This optimization problem is nontrivial for a human, even with a reasonably small number of targets and assets. In most cases, assigning a support mission implies the acceptance of increased risk to the supporting unit. Direct fire support assets must approach relatively close to the threats in order to engage them. Indirect fire support assets risk counterbattery radar detection each time they fire.

2. Integrating Fire Support

We introduced the fundamental concepts of fire support, fire support planning, and suppression in Chapter I. Here, we briefly explain how fire support is integrated with maneuver in a tactical plan.

Maneuver planning is ultimately the responsibility of the commander but is primarily handled by the operations officer or plans officer, if present. A fire support officer is typically attached from the supporting artillery command to work under the operations officer. At the company level, a forward observer performs this role, working with the weapons platoon commander. The fire support available for a unit is provided by its organic assets, such as machine guns and mortars, and attached or supporting assets, such as artillery, naval gunfire, and close air support. Liaison officers from organic, attached, and supporting commands advise the fire support officer, operations officer, or plans officer on the employment of their systems.

Fire support is typically considered from the inception of each potential course of action (COA) during battle planning (see Appendix D for a description of the human

planning process). Effective maneuver planners must have a good understanding of fires capabilities to invent realistic COAs. At the battalion level and above—and the company level for deliberate operations—the intent for fires are communicated through structured essential fire support tasks (EFSTs). An EFST contains the following elements:

- Task: each task includes a keyword—usually *delay*, *disrupt*, *divert*, or *limit*—followed by some capability of an enemy unit.
- Purpose: this explains in plain language what maneuver units, or other types of units, are trying to accomplish with the support of fires.
- Method: lists the type of fire support asset, target, timing, and desired munitions to accomplish the task.
- Effects: states the level of damage believed to be required to accomplish the task. In ascending order of devastation, the options are *harass*, *suppress*, *neutralize*, or *destroy*.

Our focus is on the organic fire support assets of relatively small tactical units. Although EFSTs are used for planning a battalion’s own mortar fires, this kind of formality is not typical for direct fire support, such as machine guns or company-level mortars. Nevertheless, we can think of an implicit EFST for a typical support by fire task with language such as that of Table 2.

Table 2. EFST for a Support by Fire Task

Task	Disrupt enemy ability to engage friendly formations during movement
Purpose	To allow friendly formations to reach their objectives
Method	Direct fire (with specific targets and times)
Effects	Suppress (specific targets)

Maneuver and fire support planning proceeds together—in fact, the definition of maneuver includes the application of fires. Although maneuver is paramount, fire support is a limited resource. In some cases, fire support is incapable of the level of simultaneous support needed for a particular COA. This could lead maneuver planners to a modified

COA that takes a more sequential, rather than parallel, approach. It is often desirable to plan multiple assets against the same target such that, if one falls through, another is still able to provide the required effects.

Generally speaking, both direct and indirect fire require time for movement, setup, and possibly target adjustment, but the indirect fire units have longer lead times.

a. Direct Fire Support Tactics

To task a unit to provide direct fire support, a commander assigns it a support by fire position. This indicates the point or area to occupy, the targeted enemy unit or part of that unit, the timing for the engagement, and the purpose of the task. The canonical example is support for an assault. The support by fire unit fires on the objective of the assault, or on enemy positions that can protect it, to neutralize or suppress the enemy long enough for the assault unit to close the distance. The support by fire unit then shifts and lifts its fires to avoid the risk of fratricide. Shifting means switching to different targets that are farther away from the leading edge of the friendly assault force. Lifting means ceasing fire. Direct fire weapons usually allow much closer fires to friendly forces than indirect fires because they can be instantly adjusted from visual or verbal cues.

Direction of fire is an important consideration for direct fire weapons, which usually have a relatively flat trajectory and long, thin dispersion pattern. If the angle between the direction of support by fire and the assault axis is 90° , then the risk of friendly fire from errant rounds is minimized. If the support by fire unit can employ enfilade fire—meaning that the direction of fire is parallel to the long axis of the target's formation—then the probability of each round impacting a target is maximized. The available directions of fire are usually limited by the number of viable positions.

The choice of a support by fire position, in addition to determining the direction of fire, also determines the range to the target. Shorter ranges to the target increase accuracy and minimize dispersion, but choosing a position outside the enemy's range or fields of fire can allow the support by fire unit to engage with impunity. If enemy weapons may be in range of the support by fire position, then the cover and concealment

offered by the position is an especially significant consideration. In undulating terrain, the preference is to engage from a reverse slope.

b. Indirect Fire Support Tactics

Artillery and other ground-based indirect fire support units can deliver overwhelming amounts of firepower on accurately located targets. Indirect fire support units are usually protected from direct fire by their distance—and usually intervening terrain—from the forward line of troops. Indirect fire support units take at least a few minutes to set up or tear down a firing position due to the size of their weaponry and the precision equipment and procedures necessary to fire accurately at great range. Even though armed with self-defense weapons, these units are vulnerable to attack by more agile maneuver forces. Ideally, planners can find firing positions such that all required enemy targets are in range and protection is naturally provided by terrain and the maneuvers of friendly forces.

To support an attack, indirect fire support units must move to firing positions within range of their planned targets and then conduct emplacement procedures, unless their initial locations are already sufficient for the fire support plan. Their movement and emplacement must be completed prior to the firing time for the first planned target. The time of flight for indirect fire is much longer than for direct fire weapons, and must be accounted for.

C. AUTOMATED PLANNING FUNDAMENTALS

Here, we review some foundational discoveries in automated planning, a sub-field of AI. We begin with a generic architectural framework for planning, which we build upon in Chapter III. In the problem discussion of Section I.D.5, we note that planning by time step does not seem to be a viable approach for combat models. However, some planning implementations (discussed below) manage to use an alternative state-based formulation. For this reason, and since classical state-based planning is the traditional introduction to the planning field, we begin this section by presenting that approach. We then describe several extensions relevant to the combat modeling domain. Our novel planning algorithm, presented in Chapter IV, combines several of these extensions, but

we discuss them in their traditional categories here. We also cover search, since automated planning can be described as a search problem in a specially defined space.

1. Generic Framework for Planning

Our generic framework for planning systems applies to both the classical state-based domains and the specialized domains of combat models. A planning system contains a database of rules, which provide all of the ways that it can manipulate its conceptual space. It takes as input a planning problem, which includes both a conceptual world configuration and goals. The planning system uses its plan generator to apply rules in search of a plan that transforms the input world configuration into one in which the goals are achieved. If it can find such a plan, it outputs it; otherwise, it outputs failure. A diagram of the framework is shown in Figure 2.

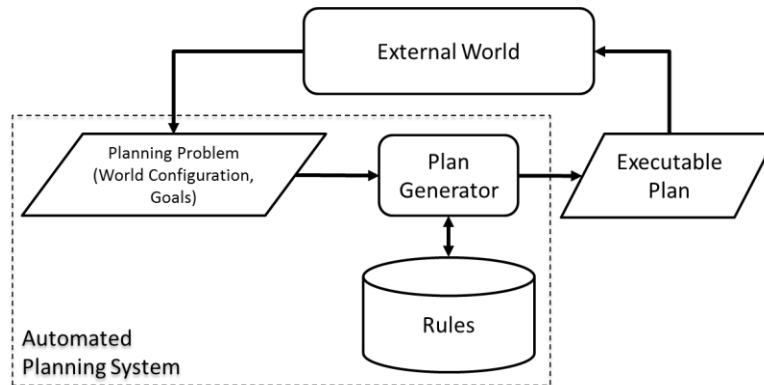


Figure 2. Generic Planning System Framework. Adapted from Ghallab, Nau, and Traverso (2004, 8–9).

The conceptual world of the planner is a model of the external world, so the effectiveness of the executable plan in achieving the goals (in the external world) is dependent upon the accuracy of the model. If the external world is stochastic, then it is probably impossible to guarantee that the executable plan will achieve the goals, but if all relevant probability distributions are known, then it may be possible to calculate the likelihood that the goals (or a subset of them) will be achieved.

The output plan is executable by a set of agents in the external world, which could be the real, physical world, or another logical system (such as an executable combat model). The precise definition of *executable plan* depends on the capabilities of the agents for which it is intended.

Some actor in the external world—a human user, or perhaps a software agent—must generate one or more planning problems and provide them as input to the planning system. The translation of the external world’s current configuration into properly formatted parameters for the planning system is just a matter of translation, but the invention of goals is usually left to human insight. A *cognitive architecture*, such as SOAR (Laird and Rosenbloom 2008), could be used to determine goals, but we consider such a capability to be outside the boundary of the planning system. The planning system may choose *subgoals*, which constitute a strategy for achieving the input goals (or other subgoals, recursively), but it is not expected to create or modify its input goals.

The simplest form of plan is a pure sequence of instructions. We assume this format in the next section.

2. Classical Planning

This section is based on the “classical representation,” as described by Ghallab, Nau, and Traverso (2004, 27–33). Classical planning is also known as STRIPS-type planning, after the popular first implementation of its approach at Stanford University (Fikes and Nilsson 1971). In classical planning, the conceptual world is defined as a finite set of ground atomic sentences (predicates whose terms are constants) with closed-world Boolean values. Classical planning excludes function symbols (including them makes planning undecidable). A world state s is the set of propositions whose values are true (the others are false due to the closed-world assumption). *Actions* change the conceptual world state by adding or removing elements from s . Only the planner can produce actions—there are no external sources of change—so the planner deals with a deterministic state transition system. Classical planning requires all of the assumptions we addressed in Section I.D.5 (remember that several of them do not hold in a typical combat model).

An *operator* is a tuple of sets (v, p^+, p^-, e^+, e^-) where v contains variable symbols and the other elements contain logical atoms, whose terms may be a mix of variables and constants. All variable symbols appearing in p^+, p^-, e^+ , and e^- must be members of v , which represents the *parameter list* of the operator. The sets p represent the *preconditions* of the operator, and its *effects* are modeled by the sets e . The superscripts $+$ and $-$ represent logical positives and negatives, which are formalized momentarily. An operator is instantiated into an action by a substitution σ of constants for the members of v , with corresponding substitutions in p^+, p^-, e^+ , and e^- . An action a (a ground instance of an operator) may be applied to a world state s only when the positive preconditions hold and none of the negative preconditions hold; that is, when $\sigma(p^+) \subseteq s$ and $\sigma(p^-) \cap s = \emptyset$. The application of a to s results in a new state s' with the positive effects added and the negative effects removed: $s' = \gamma(s, a) = s - \sigma(e^-) \cup \sigma(e^+)$. The function γ is called the *state-transition function* (Ghallab, Nau, and Traverso 2004, 28–30).

A *plan* in this representation is a sequence of actions. A *valid plan* (a_0, a_1, \dots) for a starting state s_0 is one in which the preconditions of each action hold in the state in which it is applied: a_0 's preconditions are met in s_0 , a_1 's are met in $\gamma(s_0, a_0)$, and so on. Classical planning is the problem of finding a valid plan to achieve a *goal state* from a starting state. A *goal* is a pair of sets of atoms g^+, g^- given with each particular planning problem, and a goal state is any state that contains all members of g^+ and none of g^- (Ghallab, Nau, and Traverso 2004, 21–27). The two basic approaches to solving classical planning problems are *forward* and *backward planning*. In forward planning, we produce new states from the starting state using operator instances, backtracking if necessary, until producing a goal state. In backward planning, we start with a goal state and use instances of *regression operators* (essentially the inverse of γ) until producing the starting state. In both approaches, failure is declared if all options have been exhausted without success (Ghallab, Nau, and Traverso 2004, 69–76).

For the representation just described, deciding whether a plan exists is an EXPSPACE-complete problem (Ghallab, Nau, and Traverso 2004, 57–60), which is worse than NP. Exponential complexity is common for AI problems, but practical implementations are often achievable through efficient programming techniques and heuristics. Often, the features of a domain allow heuristics and efficiencies that we cannot claim for the general classical planning problem.

3. Plan-Space Planning

In classical planning, the nodes of the search space are states of the conceptual world, and each planning step is the use of an action (either forwards or backwards, depending on the approach) to transform the state. This is termed *state-space planning*. An alternative is *plan-space planning*, in which the search nodes are called *partial plans*. These allow an incomplete specification of a plan that does not have to be a strict sequence of actions and remove the need to encode states directly during planning.

A partial plan is a tuple (W, B, L) in which W is a partially ordered set of operator instances, B is a set of binding constraints, and L is a set of causal links. W can be represented as a loop-free digraph in which each node is a classical planning operator instance, but in this case, the substitution σ for each instance is not required to bind all variables to constants.³ Each member of B is a constraint on a variable in a member of W . Each causal link in L connects a member of W with another member of W that provides one of its preconditions. The starting state and goal of a plan-space planning problem are encoded as the effects and preconditions, respectively, of special ground operators in W ordered at the beginning and end of the partial plan. We can think of planning as attempting to connect these two special operators with a network of other operators (Ghallab, Nau, and Traverso 2004, 86–90).

In plan-space planning, we need not proceed strictly forward or backward state-by-state, as in classical planning. Essentially, we may add an operator instance w anywhere between the first and last member of W . We need not immediately have all

³ Partial substitution is also possible in a lifted classical (i.e. state-space) planning implementation. See Ghallab, Nau, and Traverso (2004, 75).

preconditions for w met when we create it. All unsatisfied preconditions are called *subgoals*; they must be provided for in later planning steps. When we find that there is no possible way to meet some preconditions, we can backtrack on the choice of that operator. Once a precondition in a partial plan has been satisfied by a preceding action, we mark that dependency with a causal link. In some cases, the planner may find causal links from “collateral” providers—members of W that were originally created to meet some other subgoals. These types of causal links are usually preferred since they require fewer operator instances, but when they do not lead to a successful plan, we can still backtrack on their selection.

Causal links allow the planner to avoid inadvertently negating a precondition later in the planning process—an error that was not possible with the strictly sequential approach of state-space planning. A member of W that could break a causal link, depending on later variable substitutions, is called a *threat*.⁴ Threats can be resolved by a more restrictive partial ordering on the members of W or by more restrictive binding constraints—that is, new members of B (Ghallab, Nau, and Traverso 2004, 91–99).

In an algorithmic sense, subgoals are no different from the primary goals in the original planning problem. Each step in a plan-space planning algorithm is either the provision of a subgoal or the resolution of a threat. The strategy of which type of step to choose may be guided by generic or domain-specific heuristics (Ghallab, Nau, and Traverso 2004, 94–100).

In the language of plan-space planning, a partial plan with no subgoals or threats is a solution plan. If the external world is a fixed-step state transition system such as we described for classical planning, then we can translate a solution (W^*, B^*, L^*) in plan-space into a sequence of actions: any total ordering of W^* under a substitution consistent with B^* will do (Ghallab, Nau, and Traverso 2004, 94–100). In a deterministic world, we no longer need L^* to execute the plan exactly as given, but if changes may be necessary then we can retain the causal links to understand the impact of each action’s failure or change.

⁴ Note that our use of *threat* in Chapter IV (referring to enemy units) is unrelated to the definition here.

Alternatively, we can retain the solution (W^*, B^*, L^*) in its plan-space format for execution. This is a more natural form for worlds that do not use a fixed-step model of time—or when the time steps are so short, relative to the duration of actions, that they are not an efficient way to represent timing. Actions not constrained to occur in sequence by the partial order may occur in parallel if the action model allows this. If actions do not all have the same duration, then the plan would need to be supplemented with scheduling information in order to be executed. We explore this idea further in the discussion of continuous-time planning below.

4. Hierarchical Task Networks

The first application of hierarchy to AI planning comes from extensions to STRIPS (Fikes and Nilsson 1971) for efficiency in representation or execution. The macro operator (MACROP) (Fikes, Hart, and Nilsson 1972) groups atomic actions into a single command; allowing MACROPs within MACROPs creates a conceptual hierarchy. The ABSTRIPS algorithm uses a more powerful approach. Its *abstract operators* are made by removing preconditions from ground operators in a preprocessing step, allowing the planning system to disregard, heuristically, many combinations of actions irrelevant to the problem. In ABSTRIPS, each removal of a precondition creates a new level in the abstraction hierarchy, and the planner reaches a final solution by creating plans in each layer constrained by the plans in the layer above (Sacerdoti 1974).

Sacerdoti and other researchers evolve these ideas into Hierarchical Task Network (HTN) planning, an approach that allows domain authors to explicitly define abstract tasks that can be expanded in different ways into more finely grained tasks. These expansions, or *decompositions*, ultimately result in a set of actions with relative timing constraints—that is, a plan format similar to that of plan-space planning.

HTN planning uses the same types of operators and actions as classical planning. It additionally defines a more abstract class of logical objects: the *task*. A task has the form $t(y_0, y_1, \dots)$, in which t is a unique symbol called the *task name* and each y_i is a term called a *task parameter*. Any operator may be classified as a task by giving it a

name o and writing its parameter list members v_0, v_1, \dots as the task parameters, as in $o(v_0, v_1, \dots)$. All tasks that are not operators are called *nonprimitive tasks*. The key capability of HTN planning is the decomposition of nonprimitive tasks into other tasks. By itself, a nonprimitive task has no other formal features, but it is helpful to provide comments for users describing the abstract objectives that each is meant to accomplish. A *task instance* is a copy of a task under a substitution σ , i.e. $\sigma(t(y_0, y_1, \dots))$. For tasks that are operators, a task instance is the same as an operator instance, except that it has a name and parameters.

To conduct HTN planning, we must also define a finite set T of *constraint types*. Members of T are predicates with particular meanings whose variables may refer to tasks and logical atoms. This is a generalization of concepts such as preconditions (as in operators) and ordering (as in the partial order of a partial plan's operator instances). A *constraint instance* is a copy of a constraint type under a substitution. The planning domain in which T contains only preconditions and ordering is called *simple task network (STN) planning*. This and other types of HTN domains are further detailed by Ghallab, Nau, and Traverso (2004, 244–245).

A *task network* is a pair (U, C) in which U is a set of task instances and C is a set of constraint instances. Task networks are analogous to the partial plans of plan-space planning. Every planning branch takes a task network (U, C) and produces a new task network (U', C') by decomposing a task in U along with any of its occurrences in members of C (Ghallab, Nau, and Traverso 2004, 244–250).

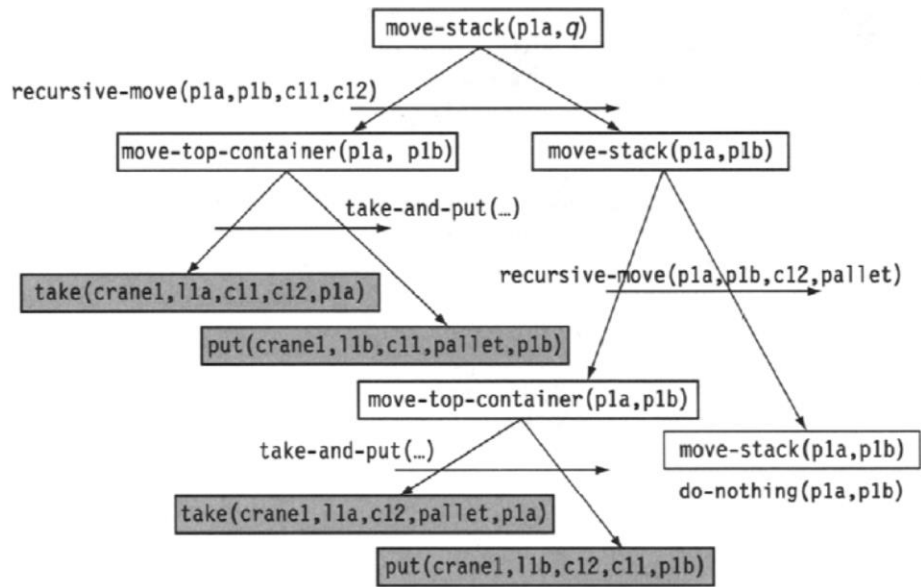
Methods describe how to decompose tasks. The finite set M contains all methods available for planning. A method $m \in M$ is a pair (w, W) , where $w = t_w(y_0, y_1, \dots)$ is a task instance and $W = (W_U, W_C)$ is a task network. Given a task instance $x = t_x(z_0, z_1, \dots)$ in task network $X = (X_U, X_C)$ such that x is unifiable with w by substitution σ , the method $m = (w, W)$ can be used to decompose X into X' as follows:

$X' = (X_U - \{x\} \cup \sigma(W_U), \Delta(X_C, x, \sigma(W_U) \cup \sigma(W_C)))$. Essentially, m replaces x with $\sigma(W_U)$ wherever it occurs in X . The special function $\Delta(X_C, x, \sigma(W_U))$ replaces x with $\sigma(W_U)$ in each constraint of X_C as appropriate for the logic of each constraint type (Ghallab, Nau, and Traverso 2004, 245–247).

HTN planning systems may also include *critic functions*, which can evaluate or make changes to task networks beyond the scope of methods. This is useful for domains that lend themselves to special “shortcut” task network transformations or when numerical calculations are needed for heuristic guidance (Ghallab, Nau, and Traverso 2004, 249–250).

An HTN planning problem is stated as a conceptual world configuration together with a task network. A solution is any task network that can be produced by the successive application of methods in M , in which all tasks are operators that can be instantiated into executable actions. This differs from classical planning, in which the problem includes goals that can be tested in the final state. However, classical-type problems with goal predicates can be expressed with task networks: a nonprimitive task with only one applicable method, whose constraints match the desired goals, may be included in the task network of the planning problem input. HTN planning also differs from classical planning due to the requirement to write methods. An HTN planner will only produce plans through decompositions described by methods in M , whereas a classical planner will try all valid noncyclic sequences of operator instances. For carefully written methods, this can significantly reduce the search space and speed up the planning process. However, HTN planning is actually more expressive than classical planning: it allows the specification of planning problems that cannot be expressed with classical planning. This expressiveness causes HTN planning, in general, to be recursively enumerable but not decidable, even for conceptual worlds with a finite number of states (Erol, Hendler, and Nau 1996). For many planning domains, the theoretical possibility of running forever is not a practical concern, and HTN planning often demonstrates better performance than classical planning (Ghallab, Nau, and Traverso 2004, 259).

HTN planning may be applied to classical state transition systems, or it may use a less rigid timing specification through partial ordering. Causal links are not typically encoded in the constraints of a task network, but a similarly useful data structure called a *decomposition tree* may be retained with the solution. A decomposition tree represents all of the task decompositions that occurred in the planning branches that produced the solution. The tree includes a copy of every nonprimitive task instance that was decomposed, with directed edges to the task instances that replaced it (Ghallab, Nau, and Traverso 2004, 248). This provides an informative hierarchical organization (Figure 3) that can improve human comprehension and localized plan repair.



In this example (which is not from a combat modeling domain), the signatures inside the boxes are task instances, and the signatures near the downward arrows represent the methods used for each decomposition. Dark boxes indicate operator instances, and white boxes are for nonprimitive tasks. Horizontal arrows represent ordering.

Figure 3. An HTN Decomposition Tree. Source: Ghallab, Nau, and Traverso (2004).

5. State-Variable Representation

An alternative to the model of a world state as a collection of logical atoms is to use a set $\{s_0, s_1, \dots\}$ of state variables, where each element s_i varies over an object

domain D_i (Ghallab, Nau, and Traverso 2004, 41–42). Logical atoms are given the object domain $\{0,1\}$, but larger domains may be used. Complexity analyses such as the ones we quote above rely on each object domain being finite, implying that there are still a finite number of world states.

Each state variable s_i is actually a function of time: $s_i(t) \in D_i$. The argument t represents a moment in the plan execution. The domain of t depends on the time representation of the conceptual world; for classical planning, we could use the nonnegative integers as state indices. At time t , the state of the world is given by the complete set of state variables $\{s_0(t), s_1(t), \dots\}$. Since every state variable is a function of time, none can take on more than one simultaneous value. This can reduce the size of the operator specification, since setting a state variable to a particular value (as an effect for the next state) implicitly negates all other possible values for that state variable at that time (Ghallab, Nau, and Traverso 2004, 41–44).

We can extend the state-variable representation by using multiple arguments for each state variable. In this case, we have state variables $s_i : D_{i,0}, D_{i,1}, \dots, D_{i,k}, T \rightarrow D_i$, where each $D_{i,j}$ is a finite set of objects in the conceptual world and T is the domain of possible time values. This format can reduce the size of the state representation if default values are implicitly recorded and most of the state variables $s_i(x_{i,0}, x_{i,1}, \dots, t)$ have default values (we need only store the non-default values). State-variable representation does not improve on the upper complexity bound of classical planning, but it can speed up the task of defining a planning language (Ghallab, Nau, and Traverso 2004, 41–44). State-variable representation is also useful for continuous-time planning, which we cover next.

6. Continuous-Time Planning

If the implicit model of time used in classical planning is too unwieldy to produce efficient executable plans, then we can consider using a conceptual world in which each operator has a start and end time represented by a pair (t_0, t_1) where $t_0 \in \mathbb{R}$ and $t_0 \leq t_1$.

One approach to planning with this time model uses the *chronicle* data structure, which partially specifies a set of *timelines*. Conceptually, a timeline is a state variable, as defined in the previous section. We can think of timelines as piecewise constant functions. During planning, the value of each timeline may be undetermined for some intervals of t . Each branch in the planning process adds or extends some of the timeline value specifications, consuming portions of unspecified intervals (Ghallab, Nau, and Traverso 2004, 326–327).

A chronicle is a tuple (S, C) in which S is a *timeline specification* and C is a set of constraints on the terms in S . Timeline specifications use a different notation than state variables. The specified values of S are recorded as *events* and *persistence conditions*. An event $s(y, z, t)$ is a change to a different value at a point on a timeline, where t is the time point at which the value of timeline (state variable) s changes from y to z . A persistence condition $s(y, t_0, t_1)$ is an assertion of a particular value y for a timeline s during a time interval $[t_0, t_1]$. If the state variable for the timeline has other parameters, such as domain objects, they appear before the y term—for example, $s(x_0, x_1, y, z, t)$ or $s(x_0, x_1, y, t_0, t_1)$. The parts of each timeline not specified by S are available for planning (Ghallab, Nau, and Traverso 2004, 326–330).

Chronicle planning is similar to plan-space planning. The partial ordering and causal links, which we saw in the partial plan structure, can be found in the constraints of a chronicle. The constraints in C are predicates using terms from the elements of the timeline specification. C may include equality and inequality assertions on time variables. For example, two persistence conditions $s(a, t_0, t_1)$ and $s(b, t_2, t_3)$ could be temporally ordered by adding the constraint $t_1 \leq t_2$; or two events $u(c, d, t_4)$ and $v(e, f, t_5)$ could be forced to occur simultaneously with the constraint $t_4 = t_5$ (Ghallab, Nau, and Traverso 2004, 330–332).

Chronicle planning uses the following concepts (Ghallab, Nau, and Traverso 2004, 331–336).

- *Consistent*: the property of a chronicle that is sufficiently constrained such that none of its timelines can take on two values simultaneously by any possible variable substitution. Consistency upholds the definition of timelines as state variables, which are functions with respect to time.
- *Establish*: to set a timeline to a particular value at a time point. This is related to classical planning's operator preconditions, which must be asserted before the operator can be applied. "Preconditions" are recorded in chronicles as constraints.
- *Persist*: where a timeline holds a particular value until a given time point. This is related to plan-space planning's causal links: an operator instance's preconditions, once established, only allow the operator to be invoked if they hold until the moment of execution.
- *Enabler*: the combination of a candidate new event or persistence condition n for a given chronicle, an additional "causal link" persistence condition l , and a set of constraints. l ensures that the required value of n (y in the event and persistence condition definitions), which must already be established in the chronicle, persists until the moment of n 's occurrence. The constraints bind the required value for n to the value of l , order l immediately before n , and ensure the consistency of the chronicle under the addition of n and l .
- *Support*: conceptually, we can only add to a chronicle's timeline specification when doing so retains its consistency. If so, the chronicle is said to support the addition.
- *Support for an event or persistence condition*: a consistent chronicle supports a candidate new event or persistence condition n only if an enabler for n exists. If so, then the "precondition" for n must already be established in the chronicle (external to the enabler).
- *Support for a timeline specification*: A consistent chronicle supports a candidate additional timeline specification (a set of events and persistence conditions) if each candidate element is supported by a combination of the original chronicle and an enabler for each of its siblings, and the combination of all such enablers can be added to the original chronicle consistently.
- *Support for another chronicle*: A consistent chronicle (S, C) supports another consistent chronicle (S', C') if it supports S' and a combination of all members of C' with an enabler of S' can be consistently added to (S, C) .
- *Plan or partial plan*: a consistent chronicle.

- *Operator*: a consistent chronicle. An operator uses timelines and constraints to specify how the world changes (or remains the same) with as the result of a conceptual activity. The use of variables in the operator definition helps in writing a domain definition concisely.
- *Operator instance*: a consistent copy of an operator under a substitution—which is a consistent chronicle.

An operator instance may be applied to a plan if the plan supports it (a case of a chronicle supporting another chronicle). Since there can be more than one enabler for each supported chronicle, there can be multiple planning branches from a single choice of an operator instance. A planning problem is a pair representing the initial chronicle and the goal chronicle: $((S_0, C_0), (S_g, C_g))$. A solution is defined as a plan resulting from the application of operator instances to (S_0, C_0) that entails all elements of S_g and C_g (Ghallab, Nau, and Traverso 2004, 331–336).

Continuous-time planning can be accomplished without specifying the durations of operator instances—in other words, by not replacing time variables with constants. This results in a logical flow of activity but does not specify how long it would take to execute the plan. If waiting is allowed—that is, persistence conditions do not have maximum durations and events do not have a “no later than” requirement—then such a plan should be executable. Essentially, each step in every piecewise constant timeline takes as long as required, and each timeline remains at its current value until it can proceed without violating the constraints. If waiting is not allowed in some cases, this planning formulation does not have enough power to guarantee that a plan is executable. A maximum duration for a persistence condition $s(y, t_0, t_1)$ would need to be specified with arithmetic, as in $t_1 - t_0 \leq \tau$. If we can also calculate the minimum duration of some persistence conditions—for example, by considering the maximum speed and distance of a route represented by a “traversing route k ” persistence condition—then we can eliminate some useless planning branches.

The chronicle planning technique’s goal specification is more expressive than that of classical planning because the goal chronicle may assert timeline values at any time points or intervals, not just at the end of the last operator instance. Another benefit of this

approach is its ability to handle combinations of interfering operator instances (Ghallab, Nau, and Traverso 2004, 336). However, extended goal evaluation, such as a real-valued objective function, is not possible without additional features. In the above definitions, success is binary with respect to whether the goal's timeline values are achieved at certain time points or during certain intervals.

7. Planning as Search

The formalization of a planning domain allows us to view automated planning as an AI search problem. Search involves five components: search nodes, a successor function, a cost function, a goal test, and a search strategy (Russell and Norvig 2010, sec. 3.3). We explain how each of these applies to the planning domain.

For planning, a search node is a partial specification of a plan. In classical planning, it is a sequence of actions (including the case of backwards planning, in which we have a sequence from some intermediate action to the final action). In plan-space planning, it is a partial plan. In HTN planning, it is a task network. In chronicle planning, the search node is a chronicle.

The successor function uses an existing search node to produce a set of new search nodes. In classical planning, the successor function uses the set of all actions that may be applied to the current state (or the set of actions whose effects are relevant to the current state, for backwards planning) to extend the sequence of actions. In plan-space planning, it uses the set of all partial plan updates that resolve a subgoal or threat in the current partial plan to produce new partial plans. In HTN planning, new task networks are produced by the methods that can be applied to nonprimitive tasks in the current task network. In chronicle planning, the successors are all of the consistent chronicles that can be produced by adding to the current chronicle each enabler of every operator it supports.

The cost function is not always used in planning—sometimes it is enough to find a plan that achieves the goals. The simplest interpretation of cost in a state-space planning domain, such as that of classical planning, is the number of actions in the plan. If actions have different costs, then the sum of action costs in the plan is an appropriate measure. For continuous-time planning, the total elapsed time (regardless of the number

of simultaneous actions) may be of interest. There is a limitless range of plan cost functions that we can consider. In the combat modeling domain, expected casualty measures or probability of success for a set of goals may be relevant. In general, a plan's cost is not always a nondecreasing sum of step costs along a search path (although for particular cases it may be). This has implications on the search strategy (see below).

The goal test for classical planning is a check of the current state against the logical atoms of the goal of the planning problem (or, for backwards planning, a check of whether the earliest state is the starting state). For plan-space planning, these goals are already present in the final, artificial operator instance; a goal test is instead a check of whether any subgoals or threats remain in the partial plan. In HTN planning, we have achieved the goal if all nonprimitive tasks have been decomposed into executable operator instances. In chronicle planning, the goal test is a check of whether a search node chronicle entails the goal chronicle.

The search strategy is the part of the search algorithm that chooses which successor node to expand next. If we need to minimize a cost function, then the search strategy should be designed to find low-cost plans as quickly as possible. If cost is nondecreasing along all planning branches, then uniform cost search is guaranteed to find a lowest-cost solution plan before it finds any non-optimal solution plan. If we can define a nontrivial admissible heuristic cost function in addition to the nondecreasing plan cost function, then we can use A* search to improve upon the performance of uniform cost search. If finding a solution plan is an absolute requirement but cost is not nondecreasing per planning step, then breadth first search may be necessary. If cost is not a factor and many viable solutions exist, then depth first search may be a faster and less space-intensive approach to finding a (possibly nonoptimal) solution. If search speed is more important than decision (we accept that the algorithm may not find a solution if one exists), then approaches such as greedy search or beam search may be effective. The effectiveness of a search strategy is highly dependent on the features of the domain in which it is applied (Russell and Norvig 2010, ch. 3).

8. Evolutionary Planning

In some domains, evolutionary or genetic search algorithms can take advantage of randomization to search for plans. We first describe evolutionary algorithms in general, and then explain how they can be used for planning.

An evolutionary algorithm (or “stochastic beam search” [Russel and Norvig 2010, sec. 4.1.3]) is a search algorithm with the following specializations. Define an *organism* as string of symbols drawn from an alphabet Σ . *Valid* organisms are those in a language V . The cost function $c: \Sigma^* \rightarrow \mathbb{R}^+$ assigns a positive number to any organism; c is such that invalid organisms have a cost greater than or equal to a boundary value B . The aim is to find a valid organism with lowest possible cost. Our search node is a set of organisms $N = \{x_1, \dots, x_m\}$ where m is constant. The *reproduction function* $r: \Sigma^* \rightarrow \{x_1, \dots, x_k\}$ produces k new “offspring” organisms by making small modifications to the “parent” organism—operations such as changing, adding, or deleting a small number of symbols, usually in a randomized way. The successor function chooses the “best” m organisms from the union of the current node’s organisms and their offspring: $s(\{x_1, \dots, x_m\}) = \{y_1, \dots, y_m\} \subseteq \{x_1, \dots, x_m\} \cup r(x_1) \cup \dots \cup r(x_m)$. The simplest approach is to choose the organisms y_i with the smallest cost $c(y_i)$, but more complicated strategies can be used to maintain diversity. For the algorithm to work effectively, r must produce valid organisms with high probability when given a valid organism as input. Most applications accomplish this by fixing the length of the organism string and defining V such that all strings are valid. *Genetic algorithms* are similar to evolutionary algorithms, but each offspring is generated from a pair of parents rather than just one (Russel and Norvig, sec. 4.1.4).

If V is a language of goal plans, then in a basic sense we can use an evolutionary algorithm to search for the goal plan with least cost. If finding a goal plan is hard, then an evolutionary algorithm is probably not helpful. If finding goal plans is easy but finding low-cost plans is hard, then evolutionary algorithms may be effective. Domains with smooth response surfaces—those in which low-cost plans tend to be clustered together and small perturbations in the input usually result in small changes to the cost—are most

likely to benefit from this approach. Due to the randomization in most implementations, evolutionary algorithms sometimes generate surprising results, which might appear creative, clever, or erratic compared to human-created plans.

9. Extensions to Plan Representations

The simplest possible plan format is a sequence of instructions for just one agent. Extensions to this could include the following:

- Collective plan: includes instructions for multiple agents.
- Parallel plan: some actions may be performed simultaneously, either by a single agent or by multiple agents (if combined with the notion of a collective plan).
- Branch plan: the plan is a loop-free digraph of nodes, where each node is an instruction or conditional switch, and each edge points to a successor of its source node. Conditional switches may have any number of nodes as successors, but instructions have only one successor. The agent begins processing its plan at the root node, proceeding to only one successor when encountering a switch, based on the result of the conditional evaluation. A branch plan allows the planner to account for a finite number of possible future outcomes.
- Persistence plan: each instruction includes a completion check, which is a conditional statement tested at some periodic or event-based interval. The instruction is repeatedly attempted until the completion check succeeds; once it does, the agent moves on to the next instruction. The completion check may be set to automatic success for a simple sequence or set to succeed automatically after a set number of attempts or elapsed time. Persistence plans allow the planner to deal with stochastic domains where an instruction is not guaranteed to succeed with the first attempt, but they make precise timing between multiple agents more difficult.
- Backtracking plan: the plan is a rooted tree of instructions and conditional switches, similar to the branch plan. In a backtracking plan, the conditional switch may lead either to a single successor or back to a predecessor node. This allows the agent to execute an entire series of instructions any number of times. A persistence plan can be implemented with a backtracking plan by only backtracking to the immediate predecessor.
- Hierarchical plan: a specialized form of parallel plan in which an agent's instructions may be instructions or goals for other, subordinate agents. The subordinate agents may not begin their actions until the senior agent

invokes them in its own execution. If the senior agent only provides goals, then the subordinate agents may need to execute a planning process once the goals are provided. A hierarchical plan models the relationship of plans in a multi-echelon command (see Appendix C for a discussion of echelons).

Combinations of these extensions result in even more complex plan languages. When all of the listed extensions are combined (with the possible exception of the hierarchical plan, depending on convention), the result is equivalent to a behavior tree, which is described in its own section below. Since such a language includes sequence, selection, and iteration, it has the descriptive power of regular expressions. The addition of an infinite stack or random access memory to plan execution (either of which is usually accessible through the instructions available to agents) would render it equivalent to a context-free grammar or general programming language, respectively. In other words, these extensions extend automated planning to the problem of automated algorithm development, which is computationally undecidable. Military plans (created by humans) do not have this level of complexity—at most, they include just one or two branches and some implied persistence. The problem of reacting to many possible outcomes is handled by the fields of cognitive architecture (discussed below) and dynamic agent control, which are related but distinct from automated planning. Although the lines between these domains can be easily blurred, our interpretation is that planners may have high complexity but output plans should be relatively simple. As we mentioned in Chapter I, invalidated plans can be dealt with in the same way as the real world—by replanning.

D. RELATED TOPICS IN BEHAVIOR DEVELOPMENT

We now cover some problems, and known tools for dealing with them, that a planning system for a combat modeling domain is likely to encounter.

1. Movement and Pathfinding

Entity movement in combat models is typically managed through *waypoints*, which are logical objects with precise location coordinates (and perhaps other information). Typically, units (or entities) are assigned series of waypoints. Each unit

moves to its next waypoint either by a straight line or along a path that maximizes or minimizes an objective function such as time of traversal.

Since the ground-based units we are considering must move across a nontrivial terrain map to achieve their objectives, pathfinding is a requirement. Pathfinding is, in fact, a special case of automated planning for which the A* algorithm is well-established search strategy (Stout 2000). The basic approach is to discretize the walkable surface into a graph of allowed locations, where each location is interpreted as a state and each edge is an action representing movement to an adjacent location (in either direction). Distance or traversal time is used as the plan cost function, and the corresponding straight-line cost from the current search location to the goal is used as the heuristic. Because the pathfinding search is limited to a 2D (or 3D) world with a number of locations that grows linearly with its area (or volume), pathfinding does not exhibit the worst-case exponential complexity of general A* search or automated planning. This is evident from the fact that the Dijkstra (1959) algorithm can solve a shortest path problem, even without a heuristic, in polynomial time. We can use Dijkstra's algorithm, or one of its slightly faster variants, to find the best-scoring path in time that is linear in the area (or volume, in 3D) of the terrain.

Battle planning involves multiple unit formations moving to various destinations with tactical considerations for other friendly and enemy formations, including time coordination requirements. This complicates the pathfinding problem because the fastest path is not necessarily the best for any of the formations that need to move. A few extensions to A* pathfinding are useful for tackling some of these issues.

The *cooperative pathfinding* techniques presented by Silver (2005) prevent formations from occupying the same map cell simultaneously—a potential problem that is exacerbated when assault units begin to converge on an objective. Silver's solution is to extend basic A* pathfinding by adding a discrete time dimension to the state space and searching for multiple paths in tandem. For realistic combat behavior, though, we are more interested in keeping units out of each other's fields of fire (which, for an assaulting units, are arcs centered on the direction of movement) than out of each other's current positions. Real-world units sometimes do pass through one another, such as in a forward

passage of lines maneuver. Preventing individual entities from passing through each other is typically required only for visual appeal; analytical simulations may ignore the issue for computational efficiency, as the SparCraft simulator does (Churchill and Buro 2013). Cooperative pathfinding does not provide information for friendly units to work together against threats, a fundamental requirement of battle planning. Furthermore, real world tactical plans often involve periods of waiting for supporting units to get into position; it is not evident how one could encode this into a pathfinding cost function.

A richer cost function for pathfinding, accounting for factors such as enemy fields of fire and generally protected areas, can generate routes that take advantage of terrain. The specialized A* algorithm that handles this is called *tactical pathfinding* (van der Sterren 2002). The cost of an edge in a tactical pathfinding search is increased by a measure of exposure to known or potential enemy positions. By carefully scaling the balance between exposure and speed, usually by a linear combination, developers can produce routes that convincingly avoid threats. A related technique is used by Rowe to score the *danger* of a position in an urban environment based on potential threats: a “function of the number of doors, windows, and corners” from which a threat could fire on that position (Rowe 2009, 6). In this approach, the contribution of each possible threat point decreases linearly as it gets farther away. To score a path (sequence of positions), Rowe uses the average of the position danger scores, but the sum could be used as the nondecreasing cost for a pathfinding implementation. Similarly, The MECH framework is used to reason about risk to a convoy route based on parameters such as visibility (Wang et al. 2015).

The pathfinding approach of Rowe and Lewis (1989) is similar to tactical pathfinding with known threat locations. Their state space (of location graph nodes) is unique due to its use of visibility volumes, where the threats can be thought of as light sources masked by the polygons of the terrain. This produces a search space in which every location is either completely masked (in umbra) from all threats or is visible to at least one threat. This property is valid as long as none of the threats moves. Their implementation uses a discrete scoring mechanism that simply accounts for whether a

node is at all visible. Later work on aircraft attack planning uses probabilistic risk over a proposed flight plan in cost functions (Secarea and Krikorian 1990; Gu et al. 2012).

In comparison to the visibility volume approach, the size of a grid or mesh discretization (Tozour 2003) is not dependent upon the number of threats or their locations. This usually results in many locations that are partially visible by threats. Developers must decide how to deal with these, since even detecting partially threatened locations can be costly. Methods less precise but faster than shadow volume algorithms include multiple-point raycasting and the depth-map method (Darken 2008). The former uses a few points in 3D space to model a complex entity’s (or, perhaps, a unit’s) volume; the latter takes advantage of the GPU’s Z-buffer.

2. Behavior Trees

A behavior tree (Isla 2005) is a way to organize finite state machines for software agents, used primarily for video game AI. Today, several behavior tree implementations exist, each with its own variations on terminology and capability. We discuss the most common themes here. The fundamental components of a behavior tree are *nodes* and *edges*, as one would expect in a tree structure. Edges are directional, and nodes are categorized as either *conditionals* or *scripts*. A conditional evaluates the game state and selects one of the nodes on its outgoing edges to execute next, but does not alter the external world state. A script does alter the world state, causing the agent to display a type of behavior appropriate for its current situation (which the conditionals should have determined). Scripts have at most one outgoing edge. When executed, they result in a *running* or *complete* return value.

Each agent’s tree is invoked repeatedly by an execution engine according to a scheduling or event system—typically, once per graphics frame. If the previous execution resulted in a running script, that script runs again. If the script was complete, then execution proceeds to the node at the script’s outgoing edge. If it has no outgoing edge, then the tree restarts its conditional evaluation at the root node. Additionally, special *impulse* messages can be sent to behavior trees by an event system. These allow the agent

to react to unexpected situations by restarting its conditional checks, even when the last script was not complete.

The current internal state of each agent is described by its set of most recently selected conditional branches and current script. Due to the tree structure, the conditionals provide a hierarchical categorization of the scripts (see Figure 4). According to behavior tree design principles, all scripts for each type of entity are placed in a single tree, and each tree is a specialization of a parent type's tree. This approach, reminiscent of subclassing in object oriented design, helps maintain a structural similarity among related behavior trees. Additionally, a *style* may be imposed on each agent to enable only a subset of the nodes in its behavior tree and modify or restrict some of their parameters.

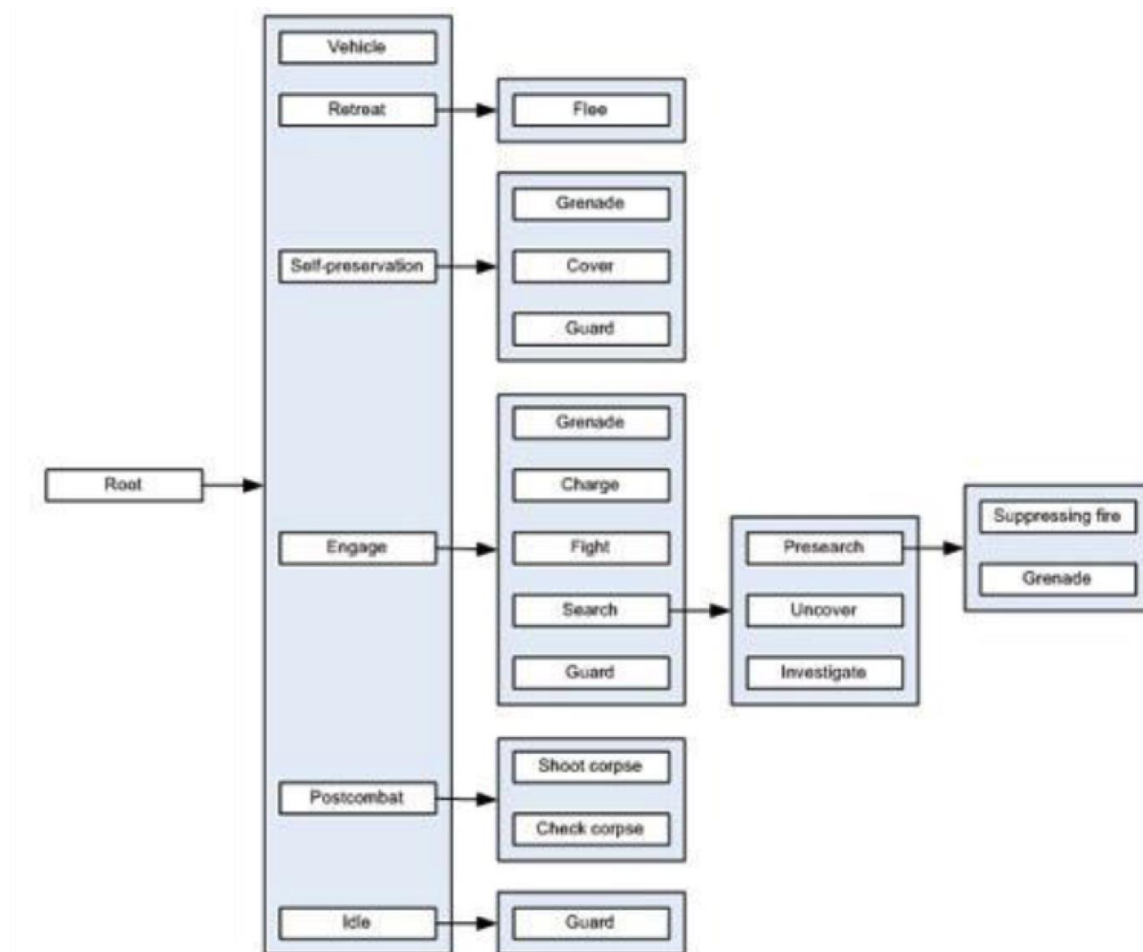


Figure 4. A Behavior Tree. Source: Isla (2005).

Specialized conditional nodes provide alternative approaches for dealing with multiple outgoing edges. *Priority* nodes evaluate each edge in a fixed order, following the first successful branch. *Sequence* nodes cause the behavior tree to move to the next edge, instead of returning to the top of the tree, when each child script is complete. *Parallel* nodes allow multiple edges to be checked and their child scripts run in the same invocation (so they appear to run simultaneously to an external observer).

Computationally, behavior trees are equivalent to nested if-then statements, but separating conditionals and scripts into named, reusable components has proven to be an effective approach to managing complexity. Although they can be used to encode plans through heavy use of sequence conditionals, Isla (2005) describes them as reactive behavior controllers for individual agents. He presents a separate *order* structure, which is similar to the operator of classical planning, for actions at the team (group of agents) level. Each order sets a squad into a particular style with customizable parameters, such as which location to occupy or which targets to engage. A plan in this scheme is a set of orders for a set of squads with some timing specification, all of which is separate from the behavior trees (but affects their configurations). The representational power of behavior trees can lead to confusion about their purpose—either a relatively fixed behavior control structure, or a situation-specific plan language. For clarity, we consider them to be the former and use the term *plan* when describing the latter, even if the plan is encoded using a behavior tree-like tool.

3. Entity and Small Unit Movement

For entity-level combat models, the tactical realism of entity and team (group of entities) movement may be of interest, particularly when humans are viewing a replication at real-time speed. At this level of detail, forward planning tends to be a waste of time because many factors can lead to the need for micro corrections. The published solutions for this challenge take the form of reactive control systems, which may pursue higher-level objectives, i.e., tasks in a battle plan. Journey (2008) explains how to use leader-follower offsets, limited-distance pathfinding, context-dependent dispersion, and randomized offsets to model natural-looking tactical formations. Reece (2003) describes

a three-tiered system in which a squad leader controls teams and team leaders control individuals, with a capability to switch dynamically between bounding⁵ and whole-formation movement at both the squad and team levels. Darken, McCue, and Guerrero (2010) describe an algorithm that accounts for additional factors present in urban environments to move team members dynamically in relation to each other while following a team-level path.

These approaches involve many entity- and team-level decisions during a replication, but few that would alter the plan at the echelons above the team or squad. One potential issue is rate of movement: a team would move more slowly while bounding compared to a single formation movement, and accounting for many danger areas in an urban corridor could result in a different rate of movement than traveling through concealment-providing vegetation. Rather than wait for a dynamic movement style decision, a planner may need to estimate the rate of movement across different regions. With this exception, we consider individual and small unit movement control separately from automated battle planning.

4. Cognitive Architectures

A cognitive architecture is an organization of AI components meant to achieve some semblance of “general intelligence,” the type of day-to-day problem solving capability that humans use to go about their lives (Laird 2012, 2–3). Planning is just one of the tools that such a system could employ. Conceptually, every entity in a combat model could use a separate instance of some cognitive architecture to choose its own goals and make its own decisions. Realistically, though, this would involve too much computation and complexity to be manageable for any scenario beyond a few entities in size. A few research projects have used cognitive architecture tools for military entity or unit control (Hill et al. 1998; Myers 1999; Lui et al. 2002; Evertsz et al. 2007), but none attempts to incorporate a truly general intelligence model. They instead use the cognitive architecture tools to implement a few domain-specific planning, goal selection, and

⁵ A tactical movement technique where part of the unit remains stationary to provide security while the remainder conducts movement.

execution capabilities. Cognitive architecture literature treats its planning component as a loosely coupled component, an approach that agrees with our individual focus on planning here.

5. Combat Outcome Prediction

To reason about the future in a combat model, we must be able to determine the outcome of units' attempts to damage or defeat one another. The traditional mathematical approach to this is to use Lanchester (1916) equations to predict, at least in a stylized model, future outcomes such as which force will win, and when, and what the final casualties will be. In the *deterministic, aimed fire* version of this model, we assume that each force is measured by two real numbers: combat efficiency and size, say α , A for one side (respectively) and β , B for the other. Each force reduces the size of its opponent continuously over time with an instantaneous rate equal to efficiency multiplied by strength: $\frac{dB}{dt} = -\alpha A$ and $\frac{dA}{dt} = -\beta B$. This means that size is a function of time, i.e. $A: [0, \infty) \rightarrow \mathbb{R}$ (likewise for B). If both sides can simultaneously damage each other, we have

$$\begin{aligned} A(\tau) &= A(0) - \beta \int_0^\tau B(t) dt \\ B(\tau) &= B(0) - \alpha \int_0^\tau A(t) dt \end{aligned}$$

Since this simple treatment allows $A(\tau)$ and $B(\tau)$ to have negative values, we call this the *naïve overkill* version of deterministic, aimed fire Lanchester equations. The naïve overkill model breaks down once either force size reaches 0. We can fix this by bounding size functions to a minimum of 0 with piecewise definitions, but this is not necessary to explain the basic concept (note, also, that we use a modified overkill model in Chapter IV). Balancing the above equations results in Lanchester's *square law*: $\beta(B(0)^2 - B(\tau)^2) = \alpha(A(0)^2 - A(\tau)^2)$. By filling in some values and solving for the others, we can make predictions (subject to the model's many assumptions) about the state of the battle at time τ , such as how many survivors remain on either side.

If only one side is able to cause damage, we can model this by setting the other side's efficiency to zero. Let us assume $\beta = 0$; then $A(\tau) = A(0) - 0$ and $B(\tau) = B(0) - \alpha \int_0^\tau A(0) dt = B(0) - \alpha A(0) \tau$. The square law collapses in this simpler case, and B simply takes casualties at a constant rate. A is unaffected.

Stochastic Lanchester equations represent the chance events of combat by adding randomness to the model. In this version, each opposing side has an integer number of troops, $N_A(t)$ and $N_B(t)$. Instead of simultaneous, continuous damage, the combat is defined as a race of exponentials: each individual troop fires killing shots at the other side according to a Poisson process with mean rates α and β . This implies that N_A, N_B are random variables, and the casualty rates are $\frac{d}{dt} N_A(t) = -\beta N_B(t)$ and $\frac{d}{dt} N_B(t) = -\alpha N_A(t)$. These are analogous to the deterministic definitions above (Billard 1979), but we cannot draw precise parallels between deterministic and stochastic models (Ancker and Gafarian 1988). For example, if $N_A(0)$ is small but nonzero, there is always some probability that $N_A(\tau) > 0$, for any finite τ . This means that the expected value $E\{N_A(\tau)\} > 0$. The deterministic equations, though, give us a finite time at which the losing side reaches size 0, so the deterministic Lanchester equations are not a precise expression for the expected value of the Stochastic Lanchester equations. However, "in situations where large numbers of units are involved on both sides, and where one of two sides is clearly stronger than the other, the number of survivors of the deterministic model is nearly the expected value of the number of survivors in the stochastic model" (Washburn and Kress 2009, 91). Modeling a stochastic combat model with deterministic Lanchester equations should be done with caution and careful testing (as we do in later chapters).

Hughes (1995) points out that, with a suppression rather than attrition model of aimed fire, the efficiency and size trade places in the deterministic square law. In this model, we assume no attrition, so A and B are just variables, not functions. We also

replace α, β with $a(t), b(t)$, representing suppression efficiency as functions of time. Each side reduces the other's suppression efficiency continuously at a rate equal to its own suppression efficiency times its size: $\frac{db}{dt} = -aA$ and $\frac{da}{dt} = -bB$, and

$$\begin{aligned} a(\tau) &= a(0) - B \int_0^\tau b(t) dt \\ b(\tau) &= b(0) - A \int_0^\tau a(t) dt \end{aligned}$$

This results in $B(b(0)^2 - b(\tau)^2) = A(a(0)^2 - a(\tau)^2)$. In other words, if suppression is the predominant effect rather than attrition (and all other assumptions of the aimed fire model are met, such as both sides being entirely within weapons range and line of sight of each other), then firepower per individual is asymptotically more important than the number of troops fielded (Hughes 1995). This is exactly the opposite intuition of the traditional, attrition-based version of the Lanchester equations. Granted, the pure suppression model relies on just as many excessively simplifying assumptions as the traditional attrition model, and this suppression is not a temporary effect as doctrine asserts, but this result is interesting nonetheless. It tells us, intuitively at least, that suppression may be just as important of a combat effect to the outcome of battles as attrition. The concept of suppression is firmly ingrained in military doctrine and practice, so it is somewhat surprising that it is not a more ubiquitous part of combat models.

Recently, RTS game AI research has pursued combat outcome prediction for purposes such as deciding whether to give battle with a detected group of units and evaluating different tactics for an engagement. Variants of Lanchester equations have proven useful here. Stanescu, Barriga, and Buro (2015) apply them to Blizzard Entertainment's StarCraft game. Since forces are made up of different unit types, their tool learns a separate efficiency value α_i for each unit type i through experimentation, then takes an average efficiency for each force (heterogeneous mix of units) to make predictive calculations. They find that an intermediate Lanchester model between the aimed fire approach (described above) and *area fire* approach has the best predictive

power. The area fire approach assumes that $\frac{dB}{dt} = -\alpha AB$ and $\frac{dA}{dt} = -\beta AB$, resulting in Lanchester's *linear law*: $\beta(B(0) - B(\tau)) = \alpha(A(0) - A(\tau))$. The intermediate approach uses $\frac{dB}{dt} = -\alpha AB^{0.5}$ and $\frac{dA}{dt} = -\beta A^{0.5} B$, giving $\beta(B(0)^{1.5} - B(\tau)^{1.5}) = \alpha(A(0)^{1.5} - A(\tau)^{1.5})$. This model can be explained by the fact that StarCraft units have maximum weapons range and cannot collocate, so large forces tend to have more units out of range, unable to contribute to attrition.

The Lanchester approach is appealing, compared to other combat prediction approaches, both due to both its preliminary experimental success and its fast computation time. The most costly step during runtime is apparently the computation of the average efficiency for a unit, which is no worse than linear in the unit size. Determination of unit type efficiencies is a nontrivial process, but it can be done offline (Stanescu, Barriga, and Buro 2015).

Alternative combat prediction techniques use simulation—that is, a simplified model of the combat model that estimate combat results over multiple time steps. Churchill and Buro (2013) are the pioneers of this approach for RTS games; their SparCraft simulator is a close approximation of the actual StarCraft combat model. Uriarte and Ontañón (2015) use more abstract model that benefits from learned data, but still requires on the order of 30 milliseconds to evaluate a potential combat—presumably much more time than a Lanchester computation (including the average efficiency determination) would require. Hinrichs et al. (2011) take a more exhaustive approach to forward reasoning in an unpredictable domain, simulating many combinations of tactical choices by both friendly and enemy and weighting the results by the probability of occurrence. This approach appears to be computationally demanding even without automated planning (they use hand-scripted COAs), but it seems to be the only one that can measure a plan's robustness to change.

6. Fire Support in Combat Models

Fire support—suppression in particular—has appeared in the literature a few times. The most relevant and well-described fire support algorithm is described in its own section below (see Cover State Graph Planning). We describe other related techniques here.

Straatman, van der Sterren, and Beij (2005) describe a technique for generating a suppression task. Their definition of suppression applies to an entity that cannot be targeted directly—i.e., for attrition fire. Instead, the suppressing entity fires at a location near the targeted entity. The shooter chooses the aimpoint by reasoning about which location the targeted entity would most prefer to occupy in order to threaten friendly entities, according to an objective function. Firing at this “ideal” location makes it appear more dangerous and therefore denies it to the target. This method is presented as a run-time behavior tool, not part of a forward planning process.

The squad and team movement controller described above (Reece 2003) has a small unit suppression feature. When the system selects the bounding form of movement for a squad or a team, it splits the unit into two formations, which alternate between movement and security. The stationary security formation may either scan for threats or, if a known threat is targetable, provide suppressive fire. The author does not provide details on how to select suppression targets or measure suppression effects. The system’s path planning is limited to “short-distance” movements of no more than 200 meters, so its suppression capabilities are, again, reactive rather than part of a planning framework.

The PlannedAssault system (van der Sterren 2014) includes a provision for calling for mortar or artillery fire support. This system is also discussed below in its own section.

7. Human Behavior Validation

DOD policy requires verification and validation for all of its models and simulations, including models of human behavior (DOD 2009a). We briefly cover the subject here to provide some initial considerations for potential future implementation, and additionally as support for experimental observations made in Chapter VI.

Unfortunately, our first point is that validation of human behavior models is a significant challenge with few useful support tools. The most common approach is face validation (DOD 2001, 6–7), which is a qualitative assessment by human subject matter experts—historically, a very small number of them. There is no established paradigm for validating an ABPS because, to date, planning has mainly been left to human designers and operators. The combat modeling community has validated *plans*, not *planning*.

Human behavior validation traditionally views the modeled human body, including the decision-making components, as the system boundary. In the framework depicted in Figure 5, everything inside the white box is part of the human behavior model. The standard assumption is that all of the components involved in the collection, processing, and output are candidates for examination. The reader may notice some similarity between Figure 5 and Figure 2, the generic planning system framework. Both constructs must take input from the external (simulated) world, use logical rules to manipulate an internal representation, and provide output. Traditionally, human representation is focused on near-term, reactive behavior. The algorithmic approaches to planning are different from those used for reactive behavior, and the periodicity of the input-output cycles are different orders of magnitude—in fact, we might want to conduct all planning before starting any replications. Furthermore, the types and formats of input and output for individual entities versus planning systems are quite different.

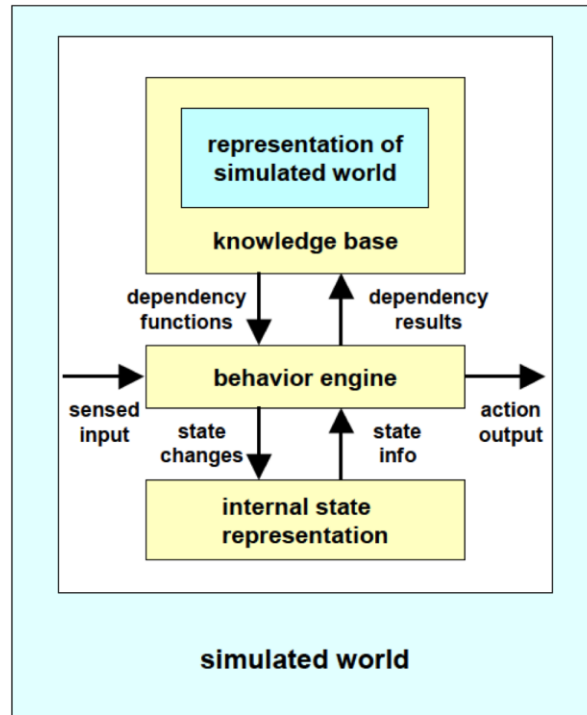


Figure 5. Human Behavior Representation Framework. Source: DOD (2001).

It appears that approaching ABPS validation as a traditional human behavior validation would be a mistake. However, many of the more abstract validation process steps and guidelines are still relevant. These include collecting requirements, determining the referent, identifying features with complex behavior, validating the knowledge base, and using an iterative approach (DOD 2001, 8–9). The planning referent, our relevant doctrine, is particularly difficult to use objectively due to its interweaving of tactical art and science (see Appendix C). This makes requirements definition even more important. The separation between the behavior engine and knowledge base, as seen in Figure 5, is analogous to the separation between a planner’s search algorithm and its rules (its database of actions, operators, nonprimitive tasks, etc.). Planning rules are different from the declarative knowledge that typically occupies a knowledge base for a modeled human. Additionally, the validation of individual rules is not enough to validate an entire ABPS. The rules themselves are likely to be structured differently than the conceptual steps or principles of human planners, as one can see from comparing almost any AI algorithm to the way a human would solve the same problem. In light of this, verification

is more appropriate for planning rules than validation. Due to the complexity inherent in combinations of many different rules, a black box approach to validation is likely warranted. Users could even consider validating individual plans when introducing an ABPS to a combat modeling process.

Ilachinski (2003, 552–555) speaks to a relationship between *emergence*, or the exhibition of recognizable behavior at some (higher) level of reasoning that is not explicitly encoded in a system, and validation. The idea here is the inverse of the usual purpose of validation: that by producing emergent, valid (with respect to a simuland) behavior in a model, we may learn something about how that behavior actually works in the real world. More specifically, if an algorithm and rule set successfully produce plans with features that human-generated plans often exhibit, then perhaps the human planners are using similar rules without conscious awareness. This view requires a belief that the human mind works at least somewhat like a computer—a fundamental assumption of cognitive science, but one that is not universally accepted by modern philosophy (Milkowski 2016).

E. RELEVANT MODELS

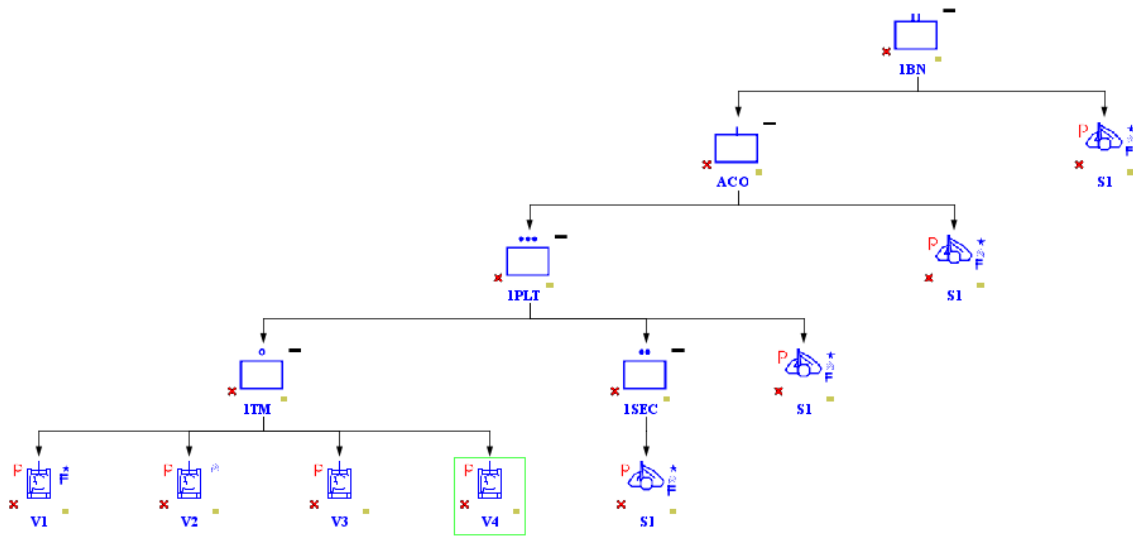
Here, we present a few of the CSEs in current use. Most of these are military analysis and training systems, but we include a few relevant games with unique features. We point out automated planning features in the systems that have them.

1. COMBATXXI

COMBATXXI (CXXI) is a stochastic, entity-level, closed-form, discrete-event model of ground combat (with some support aircraft and ships) used for analysis by the U.S. Army, U.S. Marine Corps, and a few other agencies (DOD 2015a, Balogh and Harless 2003). It is an entity-based model with a relatively high level of detail in perceptions, actions, decisions, and communications, but not to the level of first-principles physics. Entity observations are maintained separately from ground truth and are affected by models of sound, lighting, dust clouds, and skill level. Entities can only visually observe within their given field of regard, so explicit head- or turret-turning is required. User-defined behaviors may restrict entities from engaging targets based on

factors such as perceived target type. Communication is explicit: users may define messages that are passed between entities to trigger behavior. The discrete event system underlying CXXI results necessarily in a continuous model of time (to the extent of floating point representation). Space is also continuous and three-dimensional. The terrain representation is a grid of elevation postings, which defines an array of triangles using north-south, east-west, and northeast-southwest edges between adjacent postings.

CXXI has a built-in military hierarchical structure and orders (plans) system. For each scenario, the user may create units from the fire team echelon up to the brigade echelon, with command relationships similar to those described in Appendix C. Figure 6 provides an example. Every unit in a command must have one or more entities assigned to it, and each entity is customizable with respect to sensors, weapons, movement, and behaviors (the list goes on). Orders may be issued to any unit. Each order may trigger subsequent orders at its own level (allowing a sequence of orders) and at subordinate units. Orders may also trigger behaviors to cause entities to perform actions. The order writing system is not automated; it requires meticulous human input, including specification of the messages that will trigger subsequent orders.



In our terminology, this tree depicts a battalion command. The commanding unit, 1BN, contains one entity and commands one unit, ACO. At the bottom of the hierarchy, the team 1TM contains four entities (which are vehicles) and is commanded by 1PLT.

Figure 6. CXXI Command Hierarchy. Source: DOD (2015a).

In practice, the movement of units in CXXI is accomplished by assigning fixed waypoints to unit commanders. The commander proceeds from one waypoint to the next at an assigned speed, and the remainder of the unit attempts to maintain a tactical formation during the movement. A pathfinding capability with a cover-seeking feature is available but rarely used. This hesitation may be due to the battle planning process we mentioned in Chapter I, where a tactical plan is validated and then expected to remain unchanged. It is interesting to note that game developers have a similar issue trying to incorporate automated planning and learning (but usually not pathfinding): the designers lose their ability to author the player’s game experience.

CXXI includes a suppression model that uses a concept of *suppression weight*. Each time a weapon projectile impacts within a configured radius of an entity, the entity’s suppression weight variable is increased by an amount determined by the projectile type. Additionally, a future event is scheduled to decrease the entity’s suppression weight by the same amount. If an entity’s suppression weight crosses either of two threshold values—one for partial suppression, another for full suppression—then a suppression event is scheduled immediately for that entity, and one of its suppression flags is raised. Entities have no default suppression effects, but users may write behaviors that trigger on suppression events.

In CXXI, a behavior is a script for an individual entity. Behaviors are defined in Behavior Scripting Language (BSL) or Python scripts; the latter has all of the expressive power of the former and more, so we do not differentiate between them. Each behavior is invoked by a trigger event, which must be scheduled for the containing entity by the core model or by another behavior. The “Monterey Extensions” add-on package for CXXI includes a tool called Behavior Studio, which allows designers to embed Python code in a tree of conditional checks (Figure 7). The tree is invoked just like a behavior, and its contained code may invoke other trees or yield execution until awoken by a new event. This tree structure is called a hierarchical task network (HTN), but it has slightly different capabilities than the task networks of traditional HTN planning. We explain first the theoretical capabilities of CXXI HTNs, which we call *trees* from here on, then their practical uses.

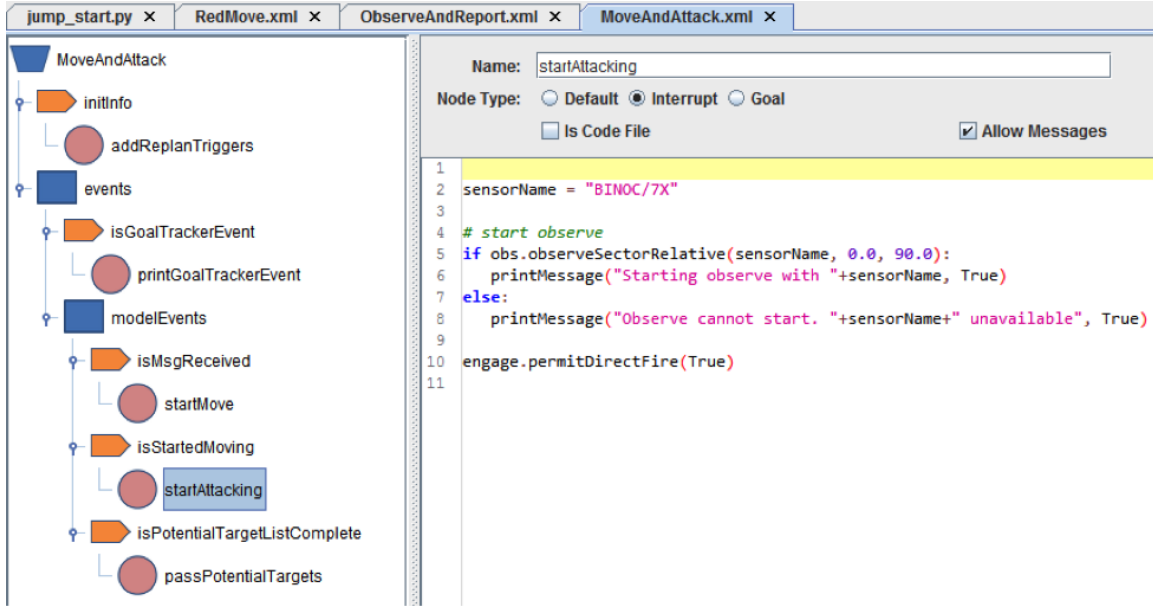


Figure 7. A CXXI HTN in Behavior Studio. Source: MOVES Institute (2015).

Computationally, trees are capable of a limited form of STN planning during CXXI replications. Behavior Studio does not have a separate structure for methods (as described above for HTN planning), so all decomposition choices must be embedded in the conditional nodes of a tree. A decomposition is accomplished by invoking another tree. The tree language (the node types and connections available in Behavior Studio) includes node sequencing but not partial ordering, so the system can only generate totally ordered task networks per entity.

If we attempt to encode an HTN planning problem using Behavior Studio, we encounter a few limitations. First, the available functions for testing the state of the executable model only check against the current, running state. To check the preconditions of a candidate sequence of operator instances, we would need a new data structure to represent the expected future state of the model. If we instead use the actual state of the model for precondition checks, yielding execution from the current tree to the event system each time we need to move to the next operator instance, we lose all backtracking points at previous states. In this “current state” approach, we would need to implement nondeterminism through many replications of the scenario. An alternative to

state-based checking is to work in plan-space (see the PlannedAssault system description below), but we would need to create a plan-space data structure with custom code.

Second, although trees have an imperative programming-style parameter-passing feature, there is no built-in way to try different variable bindings. We would need to embed code to cycle through the different possible bindings. Additionally, we would be limited to a ground-variable rather than a lifted strategy unless we made some significant upgrades. None of this is out of reach theoretically, since each node may contain arbitrary Python code, but our interest is in the capabilities inherent in Behavior Studio. Setting aside the idea of an HTN planning system, the tool is a capable development platform when we interpret its trees as either behavior trees or plans.

Used as a behavior tree, Behavior Studio trees provide reactive control of entities or units. All allowed action states can be collected in a single, organized, and maintainable structure. To promote modularity of action code (called script nodes in our earlier discussion of behavior trees), each leaf in a tree may be an invocation of a smaller tree with specific instructions to affect the external world. This provides a level of reusability similar to that of the script nodes of behavior trees. Unit control is achieved through a tree on its commanding entity. The relationship between behavior trees and Behavior Studio capabilities is further explored by Miller (2016).

Recall that a plan, when extended with certain features, becomes equivalent to a behavior tree. If we interpret a tree as a plan, then sequential nodes can specify sequential tasks. Coordination between units at different echelons can be achieved with different trees for each unit. A top-level *coordination tree* can be assigned to a command entity separate from the combat activity. Unit commanders can report their progress through message events, allowing the command entity to detect when an abstract task involving several units has been accomplished. The command entity can then send other messages to cause units to advance to the next step in their part of the plan. In this approach, each coordination tree (plan) is custom-written for the desired tactics of the scenario; it is not automatically generated. In effect, Behavior Studio provides designers with the *orders* feature of the behavior tree paradigm (as described by Isla [2005]).

CXXI is government off-the-shelf software; its Java source code is accessible to government agencies including the Army's Training and Doctrine Command Analysis Center, the Marine Corps Capabilities Development and Integration branch, and the Naval Postgraduate School (NPS). Due to its structure, its current analytical usage, and its unfortunately long scenario development times, CXXI may stand to gain the most from the present work.

2. One Semi-Automated Forces

One Semi-Automated Forces (OneSAF) is a combat simulation system primarily engineered to support large-scale training exercises for battalions (DOD 2015b). Its software architecture is designed to support many real-time users, including both event managers and entity controllers. In other words, the system is built under the assumption that humans, not software agents, will control the tactics at all but the small unit and entity level (essentially, this is the definition of semi-automated). OneSAF organizes its entities into a doctrinal hierarchical chain of command similar to CXXI. It also includes a variety of other unit-to-unit relationships such as fire support and supply, and low-level relationships such as laser designator-shooter teaming and trailer towing. Inter-unit communications depend on unit relationship settings.

Like CXXI, OneSAF has a built-in orders process. It uses the terms *task* for atomic actions and *mission* for collections of related tasks. This approach includes explicit phases, similar to those described in Appendix C. When creating a movement task, the user has the option to use "computer based route planning," which can avoid obstacles and find the shortest or fastest path. It apparently does not consider concealment from known or possible threats, as tactical pathfinding would prescribe. Units may be placed into a number of different tactical formations for movement or static defense.

OneSAF includes a large collection of automated unit-level behaviors that scenario designers or controllers may employ. Most of these are designed for squad or platoon-sized units. A variety of combat, support, and civilian behaviors are advertised, but we focus on the combat behaviors here. These include Ambush, Assault Buildings in

Area, Attack by Fire, Breach, Defend Battle Position, Seize Objective, and Support By Fire. Seize Objective is carried out as a frontal attack, with a single squad assaulting and all other squads taking support by fire positions on a semicircular arc around the objective point. Apparently, any other forms of maneuver would need to be specified through user-defined waypoints.

The Support By Fire behavior takes a support by fire position, a supported unit, one or more objective positions, and zero or more shift fire lines (Figure 8) as user input. It moves the assigned supporting units to the support by fire positions, orients them toward the first objective, and causes its entities to fire on visible enemy units. If shift fire lines and multiple objectives are provided, then the supporting units shift direction to the next objective each time the supported unit crosses a shift fire line. Entities automatically cease fire if their line of fire puts friendly entities at risk. Users may also provide a sequence of support by fire positions; each time a shift fire line is crossed, supporting units move to their next position. The user may configure the supporting units to fire only on the current objective; otherwise, entities use their own target selection logic.

Although the OneSAF Operators Manual describes behaviors involving “suppression” of targets, there is apparently no suppression behavior apart from attrition. For example, entities fulfilling a Support By Fire task cease firing if unable to “damage” their targets.

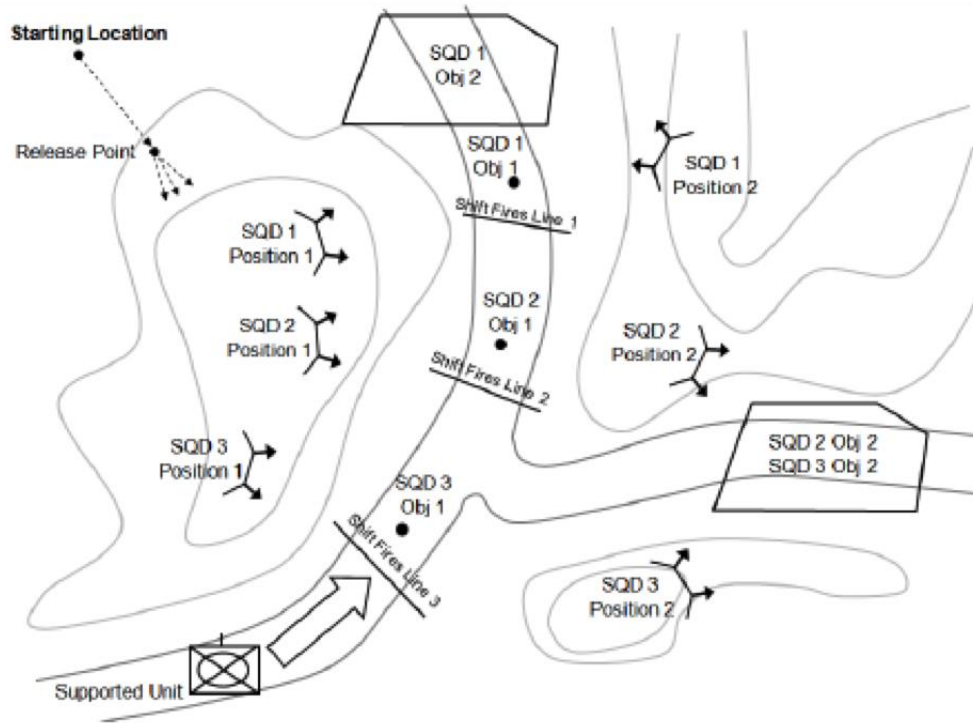


Figure 8. OneSAF Support By Fire Task Example. Source: DOD (2015c).

OneSAF source code is not normally released outside the program executive office and its support contractors. Add-on entity AI has been implemented in previous research (for example, Lui et al. 2002) by using external modules that directly control OneSAF entities with standard protocols like Distributed Interactive Simulation, or that send missions or tasks to units at a higher level of abstraction. It is a complicated tool built for real-time training events involving many human participants, and it is not a particularly good fit as a research platform. However, it appears that a more robust automated planning capability could reduce the requirement for human operators or improve some of the currently automated combat tasks.

3. Virtual Battlespace

Virtual Battlespace 3 (VBS3) is a real-time, 3D combat model with highly detailed visual and audio features (Bohemia Interactive Simulations 2015). Its physical model runs at a much higher rate than typical analytical combat models—on the order of 60 fps to align with the graphical refresh rate. Compared to OneSAF, its design is more

focused on entities and small units. VBS3 (and its predecessor, VBS2) is used as a virtual training environment for tactics and techniques, usually for platoons, squads, teams, or individuals. It shares an architecture with Bohemia’s ArmA series of military games.

A plan in VBS3 is an inherent, static part of a scenario file. It can be created or edited using a special offline interface. Plans are specified with waypoints (Figure 9). A variety of different waypoint types are available, and each triggers a specific behavior when the assigned unit reaches it—or, for human-controlled entities, provides visual cues to conduct the intended behavior. By default, units engage targets as they detect them, so even a basic “move” waypoint is enough to model an attack. Automated units use A*-based pathfinding algorithm at run-time to generate a path to each waypoint. For more configurable behavior, the “script” waypoint invokes user-defined code upon an entity’s arrival. The code must be written in the VBS scripting language.

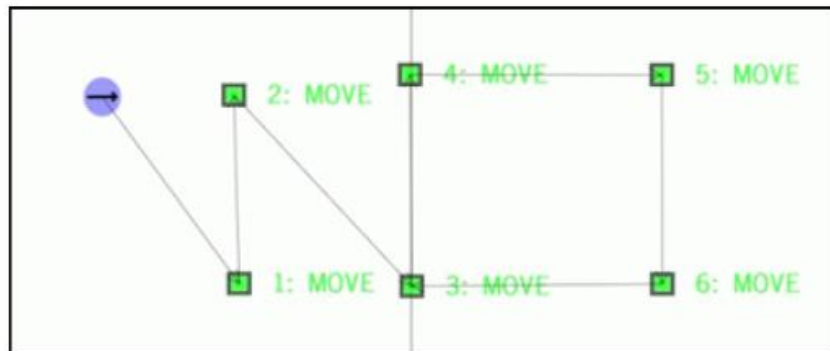


Figure 9. VBS3 Waypoints. Source: Bohemia Interactive Simulations (2015).

VBS3 includes a *suppress area* command, which instructs a unit to fire on a specified area for a set amount of time (Figure 10). This functionality is not exposed in default waypoint behavior, but it can be invoked by the `doSuppressiveFire` scripting command. Suppressive fire is more relevant in VBS3 than the other models we have discussed so far due to an interesting suppression model that, like CXXI, triggers when incoming munitions penetrate a configurable radius around an entity. Each entity’s “training” and “experience” parameters contribute to its suppression radius. Unlike CXXI, VBS3 has built-in suppression effects. When a human player is suppressed, visual

flashes and simulated flinching make weapon employment temporarily less effective. Automated entities are similarly affected, presumably by temporarily degrading accuracy or invoking a duck-and-cover behavior.



Figure 10. VBS3 Suppress Area Command. Source: Bohemia Interactive Simulations (2015).

Although the VBS3 source code is not accessible to anyone outside the Bohemia Interactive company, the richness of the VBS scripting language makes it a viable platform for receiving automated planning output. The success of the PlannedAssault system, described below, is further evidence to this.

4. Marine Air Ground Task Force Tactical Warfare Simulation

The Marine Corps uses its Marine Air Ground Task Force (MAGTF) Tactical Warfare Simulation (MTWS) to represent combat units during training events. It primarily supports battle staffs, who must coordinate the operations of large subordinate and adjacent commands. Such large formations are difficult and costly to field, so MTWS provides the surrogates. Individual entities are not modeled; units are the atomic objects. Scenario designers may create units of almost any size with types such as infantry, assault amphibian (mechanized), artillery, logistics, aircraft, and ships—the moving parts of amphibious and ground combat operations. The level of unit aggregation (the echelon at which to represent separate units) is a training design decision, depending in part on the echelon of the training audience. MTWS executes in real-time, which allows it to

operate in parallel with real world units. It can also run faster-than-real-time to “fast forward” to the next training event. It connects to Marine Corps command and control devices, allowing the battle staff to use the same tools they normally do to track and communicate with real world units. The system operators have a ground-truth view of the MTWS state, but the trainees are limited to the information they get from the training environment (DOD 2014b).

MTWS is essentially devoid of AI capabilities. Its reactive control for ground units is limited to firing on enemy units in range and stopping or regressing on assigned routes when reaching a threshold amount of damage. It has no automated planning features, depending instead on human operators to assign waypoints (DOD 2014b). Although it has highly customizable combat resolution algorithms, in practical use the combat effects are often dictated by the trainers by modifying damage attributes arbitrarily. Combat results are chosen to adhere to a script of events, which is written to attend to the training audience’s objectives. Despite this usage pattern, more automation could reduce the system operator and training support staff workload, which tend to be in high demand.

The MTWS combat model includes a binary suppression status for each unit and a fractional suppression rate, which is user-configurable. When any unit is subject to suppressing fire, its suppression status flag is raised and its firing rate is reduced by the suppression rate.

5. Map Aware Non-uniform Automata–Vector

Map Aware Non-uniform Automata–Vector (MANA-V) is “an agent-based distillation model developed by Defence Technology Agency (DTA) for use in military operations analysis studies” (Defence Technology Agency 2009, 1). The model includes a stochastic probability of hit, and it uses a fixed-length time step with a user-defined period, typically between 1 and 10 seconds. Its features include importable terrain elevation models for line of sight calculations, user-defined weapon capabilities, and floating point vector-based movement. Like all of the combat models described above, it is an entity-level (agent-level) model with entities grouped into units. In the U.S. military,

MANA is primarily used for academic and conceptual studies rather than system acquisition, real world operations analysis, or training.

The MANA behavior model uses a linear steering algorithm (see Reynolds [1999]): each agent's current movement vector is a weighted sum of the movement vectors currently influencing it. For example, an agent may be configured to move towards enemy agent type 1 with weight 2.0, away from enemy agent type 2 with weight 3.0, and towards cover with weight 1.5. A scenario designer may encode tactics through waypoint routes similar to CXXI; a velocity towards the next waypoint is just one of the component vectors for the steering algorithm. Users may define multiple states with characteristic vector weights. Due to the unpredictability of steering with multiple vectors, users tend to rely primarily on waypoints and use other vectors for micro-corrections.

The command hierarchy has two echelons: each agent is assigned to a squad, which is a group of like-modeled agents. The communications model is partially implicit: agents can see the ground-truth location and identity of everything in their field of view and (if allowed by user settings) automatically share that information between squad members or between squads. Users may define other types of more explicit communication.

MANA offers faster scenario design time than other analytical models, and it provides a wealth of output data that supports statistical analysis. It is released as a compiled Windows program, and the source code is not accessible outside DTA. Scenario files are stored in an XML-based format, so automatically generating them with a separate program is feasible—but this would have to be done without internal tools such as the line of sight calculator. Run-time replanning would be impossible without source code access.

6. RTS Games

[RTS] games...are essentially simplified military simulations. In an RTS game, a player indirectly controls many units and structures by issuing orders from an overhead perspective...in real time in order to gather resources, build an infrastructure and an army, and destroy the opposing

player's forces. The real-time aspect comes from the fact that players do not take turns, but instead may perform as many actions as they are physically able to make, while the game simulation runs at a constant frame rate. (Robertson and Watson 2014)

RTS games are combat models, at least in a basic sense. Although their purpose is entertainment rather than analysis or training, they have received much attention as a medium for AI research since 2003. As proponents explain, they occupy an intermediate position between discrete-state games like Chess and Go on one hand and the messy noise of the real world on the other (Buro 2003).

RTS AI systems are motivational for our work with combat models, but RTS games do not quite mirror the typical constructive combat simulation. The *real-time* in RTS is a bit of a misnomer; a better term would be *non-turn-based*. The distinguishing feature is that players may input commands as often as they want, and the semi-automated units carry out the commands immediately and simultaneously. Simulation time actually moves much faster than real-time, in the sense that one game involves the settlement of geographic areas, gathering of resources, construction of buildings, research of weapon technology, mustering and training of forces, and the conduct of a few battles, all in the span of 10–30 real-world minutes. Much of the planning infrastructure to handle such a domain is not relevant for the kinds of combat models we have been discussing, which are built to handle a single engagement lasting just minutes or hours of simulation time. Although tactical logistics may play a role in some scenarios for some models, RTS components such as the build-order manager or city block planner would probably just clutter a military planning framework. However, some of the combat aspects of RTS games are similar to tactical combat models in level of detail and combat resolution rules. They also seem to share the common problem of needing to develop sound plans for future actions (as human players do) in an environment with highly unpredictable future states. It is interesting to note that expert human players still outmatch the best AI agents by a vast margin (Robertson and Watson 2014), even though the RTS combat environment is entirely computational.

The RTS community uses a reasonably consistent federated problem description, broken down at the top level into “Strategy...Tactics...Reactive Control...Terrain

Analysis...[and] Intelligence Gathering” (Ontañón et al. 2013, 4). Since the field has academic attention, many advances are published in peer-reviewed journals and proceedings, complete with architectural diagrams such as those in Figure 11. Additionally, entries for most of the game-playing AI competitions are required to be open-source. This supports the combination of different solutions for various aspects of the domain. Although the specifics of RTS control architectures may not be relevant for military combat models, the concept of a shared framework is motivational. Additionally, specific discoveries for RTS AI stand a good chance of being applicable to military combat models. Here is a small sampling of successful techniques for RTS game AI that may be applicable to military combat models:

- Extracting case-based planning methods from in-game human demonstration (Ontañón et al. 2007)
- Applying Monte Carlo search to tactical planning (Chung, Buro, and Schaeffer 2005)
- Recognizing opponent plans (Kabanza et al. 2010)
- Predicting the outcome of an engagement (see the Combat Outcome Prediction section above)

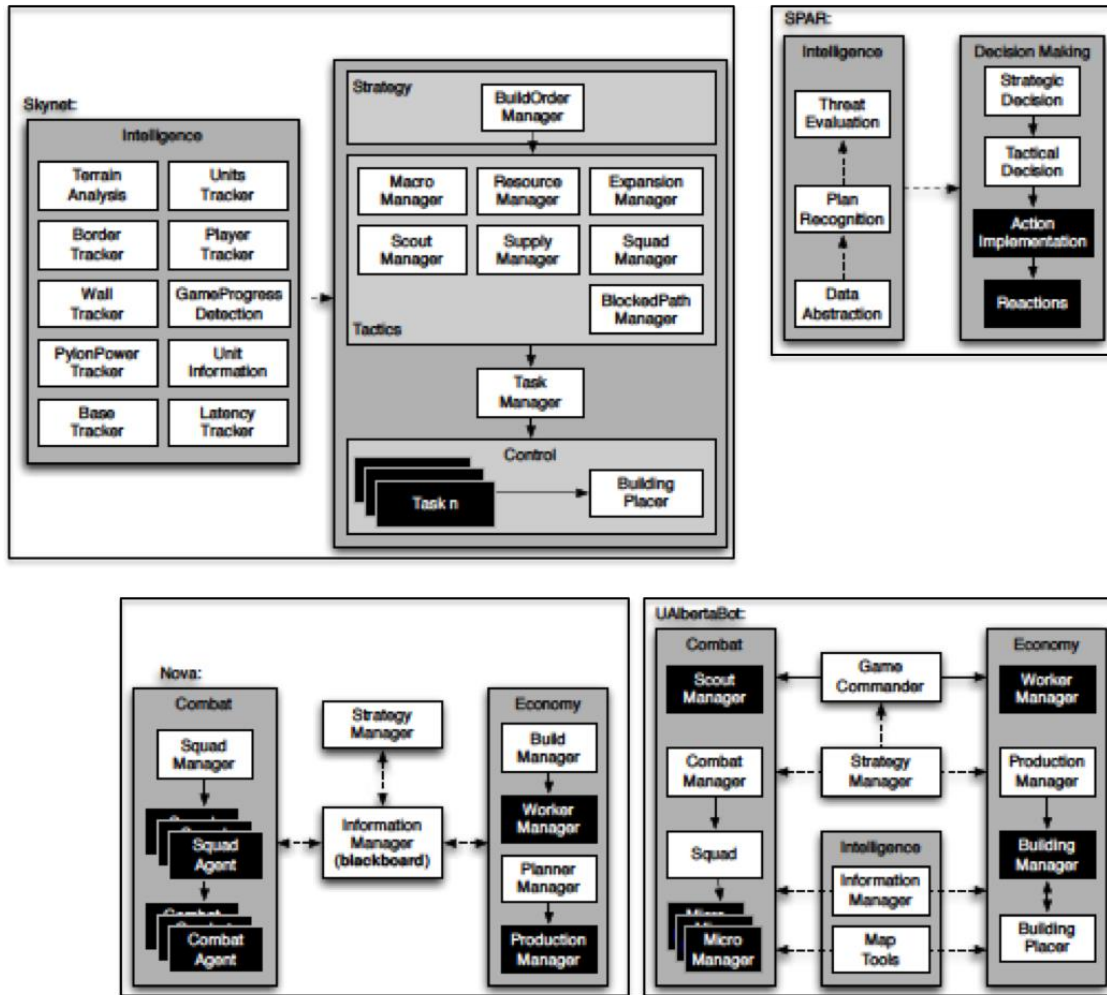


Figure 11. Four RTS Control Architectures. Adapted from Ontañón et al. (2013).

F. PLANNING IN M&S AND GAMES

In this section, we describe how some of the aforementioned planning approaches have been used in combat models. We include military-themed video games in this list; otherwise, it would be very short.

1. Goal-Oriented Action Planning

Goal-oriented action planning (GOAP) is classical planning with the state variable representation, customized for the video game domain. Its first major commercial use was as an entity action planner for F.E.A.R., a 3D infantry combat game by Monolithic

Studios (Orkin 2005). For each planning agent, GOAP represents future world states with a tuple of variables, some of which are Boolean and others that can have symbol, number, or reference types. Actions are defined for the planner as ways to change the truth value of a subset of some of these predicates; in the game, they translate to fine-grained series of actions such as moving, shooting, or grabbing objects. In addition to the normal equality check for preconditions, GOAP allows developers to insert arbitrary Boolean function calls that can check any part of the current (but not future) game state not represented in the state variable tuple.

In the F.E.A.R. implementation, GOAP searches for a plan using the A* algorithm with fixed costs per action type. The action costs are set by developers to cause entities to prefer certain types of actions to others, according to rationale like tactics or personality. The heuristic is the number of state variables not yet matching the goal, apparently scaled such that this is an admissible function.

Pittman (2008) augments GOAP with a multi-echelon military command hierarchy. In his approach, the actions at every echelon except the lowest are, in fact, goal weights for subordinate units. Each unit runs its own GOAP algorithm, using the customized state representation and action set for its echelon and the weighted goal set provided by the current action of its commanding unit. Each unit adjusts its goal weights based on local priorities, such as survival, which adds realism to the AI. This approach blends planning with cognitive architectures, and it is an alternative to an HTN planning approach to military echelon abstraction. If the state space at each level is efficiently searchable, then GOAP with command hierarchies may be a good alternative to HTN planning, with the benefit that no decomposition methods are needed.

In another example, Amstutz, Andra, and Rice (2012) describe a GOAP-like planner implementation for a single entity in IWARS, a combat model used by the military acquisition community. IWARS scenarios typically have a very small number of entities and little or no command hierarchy, so the choice of GOAP over HTN planning is not surprising.

2. HTN Applications

The first example of HTNs applied to the military planning domain is SOCAP, an application of the SIPE-2 planning system to operational-level plans (Wilkins and Desimone 1994). The hierarchies of SOCAP are task-oriented, not military echelon-oriented. Its domain objects only span one or two military echelons. In SOCAP, spatial reasoning is restricted to a highly discretized map of strategic regions and cities, as we would expect at the operational level of warfare. Tactical concepts like fields of fire and suppression cannot be naturally represented at this level of spatial and temporal abstraction. One of the conclusions of the work is a need for better temporal coordination, such as actions that need to occur simultaneously—an essential aspect of tactical planning.

The use of HTNs for game AI begins with small teams of entities—for example, online coordinated strategy selection and control in a first-person shooter (Hoang, Lee-Urban, and Muñoz-Avila 2005) and offline action planning (Gorniak and Davis 2007, Kelly, Botea, and Koenig 2008). The Killzone 2 and 3 games, by Sony’s Guerilla Games studio, introduce an additional command echelon. Entities use one database of tasks and methods, and teams use another; both use a common HTN planning algorithm during runtime (Straatman et al. 2013). An additional squad controller, which assigns goals to teams, uses a custom algorithm rather than HTN planning. This approach is similar to GOAP with command hierarchies (Pittman 2008), but with HTN planning at each node instead of GOAP. As with SOCAP, the hierarchies of task decomposition are separate from the (comparatively low-level) military hierarchies.

HTN planners have been used for RTS games, as well. Muñoz-Avila and Aha (2004) describe a limited HTN planner for the open-source Stratagus game. The case-based reasoning engine of Ontañón et al. (2007) extracts methods (ways to decompose tasks) from human player demonstrations of the game Wargus. Ontañón and Buro (2015) integrate HTN planning with minimax search for the μ RTS game. Despite the appeal of controlling RTS gameplay at multiple layers of abstraction—an apparent advantage of humans over computer opponents—HTN planning is not the dominant approach for RTS research or the popular StarCraft “bot” competitions.

3. **PlannedAssault**

The PlannedAssault tool (van der Sterren 2014) generates attack and defense plans for VBS2, several variants of Arma 2, and Arma 3. A separate product from its target CSE, it takes advantage of the VBS scripting language to generate scenario files. The user selects and places units and assigns missions (to attack or defend a single objective), form of maneuver (a few variants of frontal and flanking attacks), and environmental parameters. PlannedAssault accepts up to 10 squads per side. Each squad contains about a dozen entities or less for infantry, or up to four vehicles (we refer to vehicle units as “squads” for consistency, even though a group of four vehicles is usually called a platoon). Squads are grouped into platoons⁶ with multiple platoons per competing side, so we can think of each of the two sides as a company. Although the squads can contain a few more entities than the teams in Killzone, this structure has the same number of command echelons. It differs by using a single planning agent for the entire company, by using the HTN hierarchy to reason at different military echelons, by its plan-space representation, and by doing all of the planning work prior to execution.

PlannedAssault uses a custom-built HTN plan-space planner. Each task has a fixed hierarchical level, and each method may only decompose a task into tasks with lower levels. Some levels correspond to echelons (squad or platoon). Additional levels within a single echelon are used to decompose tasks conceptually into sequential steps or parallel efforts for sibling commands. For example, in Figure 12, the “team” (platoon) level assigns roles to platoon-sized units, and the “tactics” level synchronizes unit actions once roles have been assigned. This strict task hierarchy serves a useful software engineering purpose: errors in the plan can be isolated by determining the level at which the problematic behavior is defined. This approach is analogous to the segregation of duties in an internetworking protocol stack.

⁶ The author calls groups of squads “teams,” but we call them “platoons” here for consistency throughout this document.

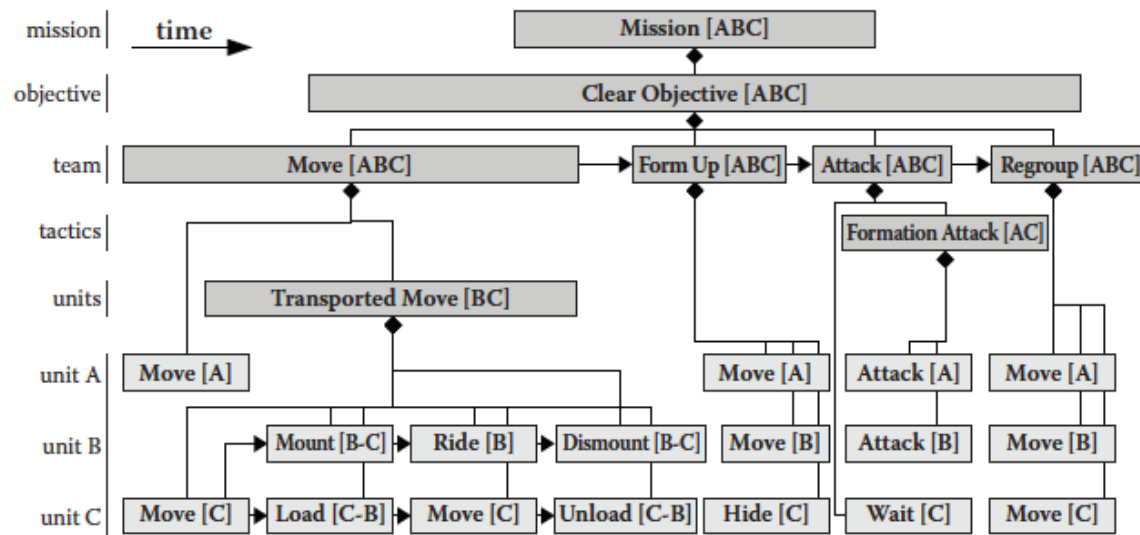


Figure 12. A Partial Plan in the PlannedAssault Representation.
Source: van der Sterren (2013).

PlannedAssault uses the general approach of decomposing at higher levels of abstraction first to avoid detailed expansion of partial plans that are clearly useless. However, these “temporarily ignored” partial plans are kept around at various backtracking points. The algorithm will return to them if the initially promising options become less attractive upon further expansion. This approach differs from the human planning process. In doctrinal military planning (see Appendix C), each headquarters makes judgment calls about how decisions at its own level will translate into good options for all lower echelons, using knowledge and insight gained from working at those echelons in the past. PlannedAssault does not need to make these kinds of guesses—it can propagate many different high-level tactics down the command tree, holding its final decision until the lower echelons have found their best possible plans for each. This approach would be intractable and mind numbing for human planning teams, but it is a good fit for a computer system. Even for a computer, this approach can generate a huge number of planning branches, so PlannedAssault limits variable binding options to just a few combinations. Its *advisor functions* (another term for the critic functions of traditional HTN planning) provide these bindings using spatial and temporal calculations such as tactical pathfinding.

The PlannedAssault search algorithm is a variant of A*, using minimum plan duration as the heuristic. Other factors such as path exposure (the opposite of concealment) contribute to step cost function. This approach has the nice property of examining aggressive plans first but checking for more tactically feasible options before committing to an answer. It is important to note that, although one can define a numerically consistent heuristic using time, there is no guarantee (for VBS2 and similar environments) that the plan output by such an A* search actually results in the “best” performance. However, this approach is a compelling compromise between planner running time and plan quality.

PlannedAssault can generate a single pair of plans (one per competing side) in about five minutes. According to the author, the majority of computation is pathfinding and terrain reasoning, not HTN planning branches. Apparently, the attacking side has no access to details about the defending side’s unit placement. From a doctrinal standpoint, this would lead to a movement to contact operation rather than a deliberate attack—but PlannedAssault is external to the VBS2 execution engine, so adjusting plans during execution (as a movement to contact would require, once enemy locations are identified) is not possible.

4. Cover State Graph Planning

The objective of the cover state graph search algorithm (Shi and Crawfis 2014) is to find the optimal combination of movement and suppression fire for a set of units⁷ F moving from a point A to a point B. Optimality is defined in terms of damage sustained by all team members from a set E of invulnerable threat units in fixed locations. The walkable surface is discretized, and each resulting location is annotated with binary visibility and expected damage per unit time with respect to each threat. The allowed actions are to move to an adjacent location or engage a visible threat with suppressive fire. Suppression is assumed total—the target may not cause damage while being suppressed—and only effective as long as the friendly unit keeps firing.

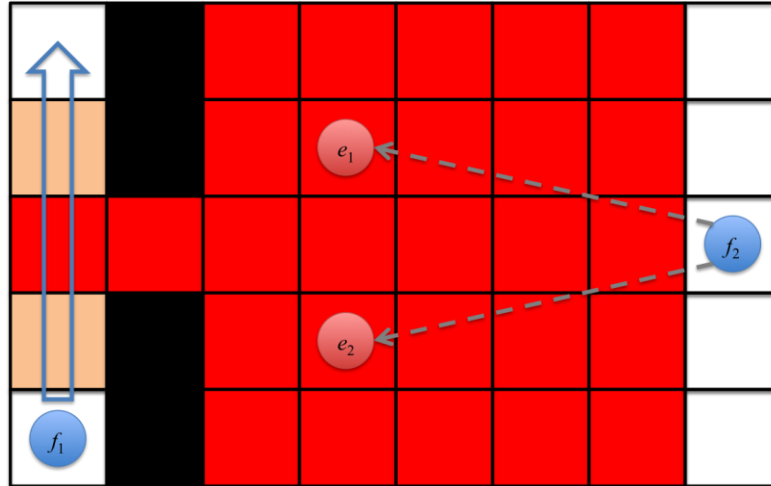
⁷ The authors speak in terms of entities (“squad members”), but the same algorithm can be applied to units that move as separate formations.

A key aspect of the authors' solution is an implicit, discrete model of time. We discuss time in terms of *micro steps* and *macro steps*. Each micro step corresponds to a unit's traversal of a single location in the discretized map. The micro time step is used when calculating the damage received by a unit traversing a sequence of locations. Before explaining the macro time step, we first need to define some other terms. The *cover set* is the subset of the discrete map locations where each member is well-protected from threat damage and is adjacent to good firing points for suppression (according to an evaluation function). A *cover state* is a one-to-one mapping from the set of friendly units to the cover set. A macro time step, then, corresponds to a transition from one cover state to another by the movement of exactly one unit, which requires one or more micro time steps. The *cover state graph* (V, M) for a search problem is the set of all cover states reachable from the start state (V) and a subset of the legal transitions between states (M), where M is filtered to disallow very long movements.

For a given transition $m \in M$ in which f , say, is the mobile unit, the authors define an optimal suppression target assignment as the one-to-one mapping $S: F - \{f\} \rightarrow E$ that minimizes the sum of damage costs to f . The damage cost is the sum of the annotated damage for each threat in each location that f traverses during m , less the damage of the suppressed threats (those selected by S). Suppression is only counted for $S(f') = e$ if f' has visibility of e .

What the authors do not point out, though, is that such a targeting selection is not necessarily optimal when considering the micro time step. Figure 13 depicts a single transition in a cover state graph for two friendly and two threat units. Friendly unit f_1 is directed to move to the cover location in the upper left, and f_2 is able to choose either threat to suppress. If f_2 can only choose one target for the macro step, then either choice results in a damage cost of 2 for f_1 's movement. If f_2 is allowed to suppress e_1 for two micro steps and then suppress e_2 , it can reduce the damage to 1. Considering this level of detail would increase the size of the search space considerably, but it seems reasonable to

expect a fire support algorithm to switch targets when a suppression target ceases contributing to the reduction of risk.



Black squares block line of sight. Red squares are targetable by both red threats, and tan squares are targetable by only one. The damage cost is 1 per location per targeting threat per unit time.

Figure 13. Suppression Targeting Example

A few other restrictions of this approach make it difficult to employ in accordance with real world tactics. The most limiting aspect is the assumption that only one unit will move at a time. Relaxing this would increase the number of edges in the cover state graph. Furthermore, if multiple units can move simultaneously over paths of different length, then some will reach cover before others. We may then wish to begin moving the earlier-arriving units to their next cover positions without waiting for the slower units in order to keep pressure on the enemy commander; this approach would severely complicate the nodes and edges of the cover state graph.

To model an attack operation, we would need to introduce a concept of assault units whose geographic objectives are the threats themselves. Even if suppression fire cannot destroy the threats, we would expect that assault units of sufficient size, and with effective supporting fires, could eliminate them (otherwise we would not be attacking). After a threat is eliminated, the algorithm needs to ignore its damage scores; otherwise, units will plan to suppress threats that no longer exist.

Although fixed costs per location make pathfinding more efficient, if units can take different routes across each discrete location in their path (cutting corners, for example), it may be desirable to work with a more refined damage cost calculation that accounts for the precise time and distance from threats along the detailed path. We also note that the authors benefit from a significant representational efficiency by assuming all friendly units are equivalent, but this is not a valid assumption in most combat models.

A distinct advantage of the cover state graph approach is that it supports A* search. The step cost function for an edge in the cover state graph is the expected damage to the moving unit after suppressed targets have been discounted. The heuristic function needs to underestimate the cost for all units to reach location B from their current location. The approach to this is to assume that each of the k friendly units will follow a least-cost path with the $k - 1$ most damaging threats suppressed (by all other units) at each step. It may not actually be possible to achieve this level of suppression with every unit following its shortest path, but a lower cost is not possible, so it is an admissible heuristic.

5. Command Ops 2

Command Ops 2 (CO2) is the engine for a series of World War II historical computer wargames produced by Panther Games (2015). It is essentially a constructive, time-stepped combat simulation with units modeled from the division down to the company (for line units) and platoon (for some support units) level, similar in level of detail to the MTWS system described above. The map is discretized into 100×100 meter squares with annotations for elevation (affecting visibility) and terrain type (affecting visibility and movement). In the typical game scenario, the player must seize a number of objectives using a hierarchically organized force (Figure 14), while the computer AI opposes with a similar force.

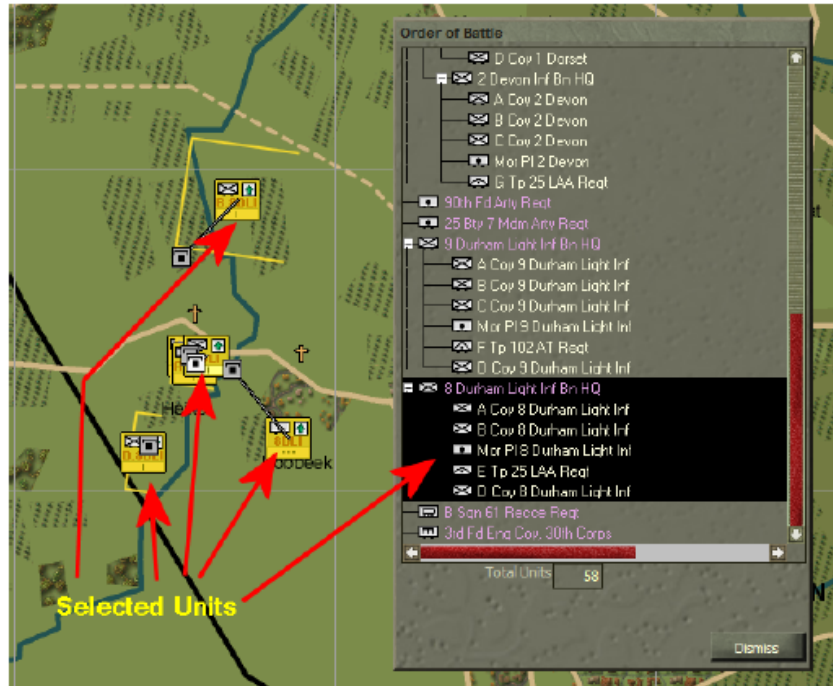


Figure 14. CO2 Hierarchical Order of Battle. Source: O'Connor (2015).

CO2 is unique in the video game space due to its AI's ability to control multiple echelons of aggregate military units realistically and reliably while accounting for terrain and threats. The AI controls not only the opposing force, but friendly units as well. It handles most of the detailed planning to carry out the high-level commands issued by the player. For example, if the player assigns a tank regiment to attack a location held by an enemy unit, the AI will select multiple attack positions, objectives, and routes for its subordinate battalions and work out the timing for a coordinated attack. It will then do the same for the battalions' subordinate companies (Figure 15). During execution, the AI moves its units in column formation to the attack positions, orders them into on-line formation, and conducts the assaults on the objectives. Supporting artillery and mortar units answer calls for fire from the engaged units as well. If an attacker suffers significant losses, the AI automatically plans a retrograde and attempts to complete the attack with the remaining forces. This is exactly the type of behavior that combat models need, and which they currently lack.

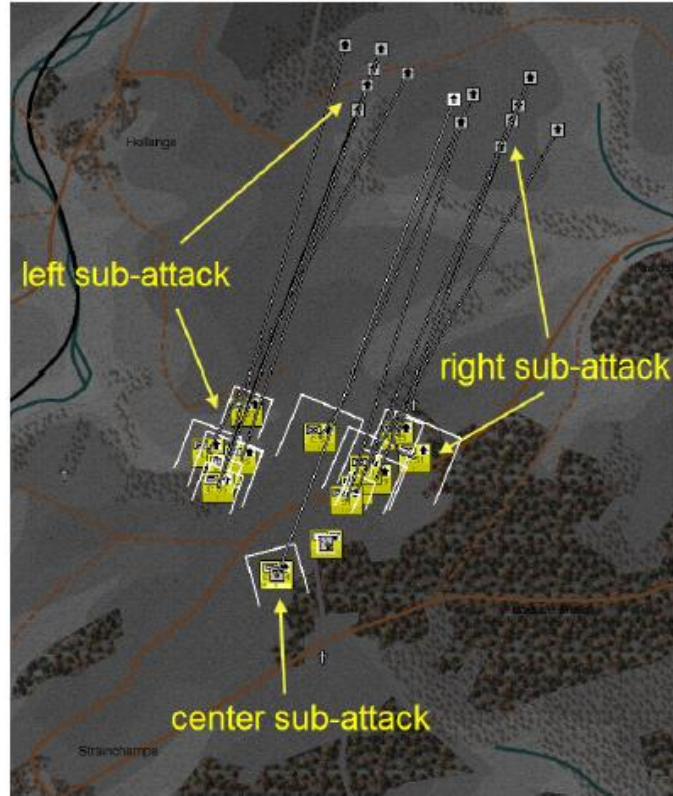


Figure 15. CO2 Coordinated Attack. Source: O'Connor (2015).

Of course, CO2's internal design and source code are proprietary, so we cannot compare its internal representations and algorithms against other approaches or use them as a basis for further research. However, its demonstrated realism, adaptability, and dependability with no problematic computational delay indicates that automated battle planning is not out of reach for combat models.

6. Evolutionary Battle Planners

Evolutionary algorithms have been used for maneuver planning in a few interesting cases. The Battlespace Terrain Reasoning Awareness Battle Command (BTRA-BC) Battle Engine (BBE) (Schlabach 2010), which originally appeared as the FOX-GA algorithm (Hayes and Schlabach 1998), calls its evolutionary organisms *courses of action* (COAs), borrowing the term from the military planning process (see Appendix C). The BBE COA is a fixed-length string. Subsequences are defined as plan property specifications such as unit-to-lane assignment, combat formation per lane,

priority of effort, bypass criteria, and reserve triggering criteria. BBE allows users to define their own COAs as first-generation input, but the system still works if only simple, default COAs are used. It has both evolutionary and genetic (crossover) reproduction functions. Its cost function is computed using deterministic Lanchester equations against enemy COAs, which are part of the system input. The search algorithm includes a strategy for retaining COAs that are distinct from each other. Users may influence tactics somewhat indirectly by adjusting weights on the multifaceted COA cost function, which can cause different forms of maneuver (see Appendix C for a definition of this term) to arise in the output plans.

BBE uses the following strategy to maintain the fixed string length of its COAs. First, a terrain analysis tool discretizes the scenario map, filtering out impassable areas. It uses this representation to form a mobility graph: each connection between traversable areas is a node, and each pair of connected areas is an edge. The edges are annotated with traversal time, concealment, and other parameters. A finite number of paths from units' start points to user-selected objective points are marked as *virtual lanes*, and each is assigned a bit string. These bit strings are used in COAs to represent the geographical maneuvers of a task force (set of units). Each virtual lane is interpreted as a geographically isolated route, so combat results of each are computed independently (Schlabach 2010). BBE is meant to support real world battle planning, but it apparently has not been widely adopted.

Kewley and Embrechts (2002) use a similar approach. Their discretization of space is a grid of 300×300 meter squares. Each unit is assigned to one grid per plan phase, with a small, fixed number of phases. At initialization, each unit is assigned a completely random grid location for each phase. The plan is run through a stochastic simulation multiple times against several enemy COAs to calculate the cost function, and then the evolutionary algorithm changes some of the unit destinations in some of the phases for the next generation. Enemy plans are co-evolved with the friendly plans, so the evaluations are in some sense against the most dangerous enemy COA. The authors provide some statistical evidence that this algorithm can outperform knowledgeable human planners with respect to the simulation output. This seems at odds with the total

dominance of human players over AI in the RTS arena, but the humans in this experiment had very little time to learn the details of the combat model—at least when compared to the time investment of an RTS game enthusiast. Interestingly, the final computer-generated plans look reasonable and seem to employ different types of friendly forces in doctrinal roles, even though no such doctrine was explicitly provided to the algorithm.

Evolutionary planning algorithms have also been used in RTS games on a few occasions (Ponsen et al. 2006, Wang et al. 2016), but not in ways that map directly to the kind of tactics needed for combat models.

G. SUMMARY

Tactics and planning doctrine are described as a combination of art and science. The latter is the appropriate focus for an automated planning system, at least for the state of today’s behavioral representations. Military commands are organized with a strict hierarchy, and different kinds of reasoning are needed at different hierarchical echelons. The capabilities of today’s computer systems seem more useful—that is, able to improve upon the capabilities of human planners—at the lower tiers of the tactical level of warfare, where precise distances, timing, and the effects of terrain are more important. Human processes are designed to support the commander’s grasp of the problem and creative design, and they depend heavily on intuition and experience, so a direct translation of human planning processes to computer planning algorithms is not appropriate.

We have many representational and algorithmic variations to choose from when developing an automated planner. Classical planning is difficult to apply to the domain of tactical combat models due to its implicit representation of time and its focus on binary goal achievement. Often, the question is not whether a mission is achievable but at what cost. Nevertheless, some military-style games have found success with GOAP, a variant of classical planning.

Reasoning at different command echelons is a natural fit for HTN planning, but it is not the only demonstrated solution. Some architectures use a separate planning agent for each unit, where each output plan is provided as input for the subordinate units’

planning. This is similar to how multi-echelon planning is done in real world commands. However, a benefit of the multi-echelon HTN planning approach is the propagation and scoring of tactical choices down to the lowest echelons. By maintaining multiple backtracking points, a multi-echelon HTN planner can consider the impact of high-level choices on low-level options, a computational surrogate for the intrinsic tactical knowledge of human planners. The PlannedAssault system (van der Sterren 2014) is a demonstrated implementation of multi-echelon battle planning.

To produce useful battle plans, an ABPS needs more than just a planning algorithm. It must be capable of tactical pathfinding and combat outcome prediction. It must be able to specify movement styles and tactical formations in its plans, and it may need to communicate with cognitive architectures or behavior trees using methodologies such as orders and styles. A* search is a widely applied tool for automated planning. Although it is a ubiquitous technique for pathfinding, a number of clever techniques have been employed to align more general plan search with the requisite A* assumptions. Evolutionary algorithms are an interesting alternative approach to tactical planning at a single echelon, well-suited to the fact that plans tend to be easy to find but hard to optimize.

Once we have a functioning ABPS, validation becomes a requirement. The traditional entity-based approach to human behavior validation is not a good fit, and validating rules one-at-a-time is not sufficient. Black box testing seems to be a necessity. As a first, cautious step, we can validate individual plans, but a trusted replanning capability will require a more general validation.

Although some combat models have had suppression features for quite some time, more are beginning to appear in new products and versions. However, the cover state graph algorithm (Shi and Crawfis 2014) seems to be the only attempt to account for a suppression model during planning. Although it is effective under some restrictive assumptions, a less-constrained and more fluid fire support planning capability is needed.

An important insight of RTS game AI research is that monolithic designs do not work. All competitive RTS controllers are federated, and each component uses tools and

techniques specialized for its purpose. These components must communicate with each other; and ideally, the work from one research project can be integrated into the architecture of another. These ideas, in the context of automated planning, lead us to the next chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

III. CONCEPTUAL PLANNING FRAMEWORK

A. MOTIVATION AND ORGANIZATION

An ABPS produces plans for use in a combat model. In the previous chapter, we reviewed several examples of effective systems that fit this basic description. No single approach stands out as dominant, in part because the combat models they support have different purposes and structures. A common assumption in automated planning is that we must settle on just one computational approach for a given problem—classical planning, HTN planning, an evolutionary algorithm, etc. However, some domains are so complex that they require a federation of subsystems with different capabilities, each with a different internal design. The problem of automating RTS game playing is an example of a field closely related to battle planning where monolithic approaches do not work. Even if we restrict our attention to the planning components of an RTS bot (ignoring components like micro-unit control, building placement, scouting, and so on), we usually find multiple tools applied to different aspects of the planning problem.

It seems a reasonable assumption that the tactical combat models we reviewed in the previous chapter, and others like them, are complex enough to require heterogeneous planning solutions. This idea is quite analogous to the way human battle staffs consider the six warfighting functions in a separate but integrated way, as mentioned in Chapter I. We do not claim that an ABPS should be subdivided exactly along the same lines,⁸ but whatever the organization, the separate tools must be integrated. In software projects such as this, we often wind up with a collection of tools created by different developers for slightly different domains, each with its own data structures, modeling assumptions, and functional scope. This complicates the integration process, because now we must translate both syntax and semantics, and we may need to disable parts of each tool to avoid duplication of effort. However, loose coupling between components helps in improving or replacing each one—for example, when new capabilities are demonstrated

⁸ Designing a planner or cognitive architecture whose subcomponents are exactly aligned with the six warfighting functions is still an interesting approach to the problem, one proposed in conversation by M&S researcher and soldier LTC John Morgan.

through a research project or in a related industry. In this chapter, we offer some concepts to help organize and manage an ABPS.

Let us define some terms before proceeding. An *architecture* is the organizational scheme of a specific executable model or part thereof. Architectures delineate the major components and subcomponents of implementations and the interfaces between them. A *framework* is an abstract architecture applicable to a variety of implementations. It distills the design principles, common themes, and vocabulary of a class of architectures. This chapter describes a framework for ABPSs. Evidently, none yet exists for this specific domain—even the closely related RTS control systems display a variety of architectures.

In this chapter, we remain intentionally abstract about implementation details. The goal is to establish concepts that apply regardless of certain modeling decisions. Chapter V describes a particular architecture (for a specific executable combat model) conformant to this framework. There, we get more specific about modeling choices, but for now, we remain agnostic about the following:

- The level of aggregation: whether the smallest modeled unit is a person, vehicle, team, squad, or some larger echelon
- The types of combat units and their capabilities
- The dynamic behavior models governing small-unit or individual entity decisions
- The specific instructions that can be issued to units and composed together in a plan
- The way terrain is represented and its effect on combat outcomes (although we assume there is some kind of nontrivial terrain model)
- The mode and resolution of time advancement, such as in fixed-duration steps or by event
- The details of the combat adjudication model, which may be deterministic or stochastic

1. Top-Level Framework

Our Conceptual Planning Framework is organized at the highest layer of abstraction into Planning Data, Planning Input, and the Plan Generator. The remaining

sections of this chapter are aligned with these three major components. A summary view of the framework is presented in Figure 16, which also includes boxes for the executable plan (the output) and the corresponding CSE. The dashed line in the figure delineates the system boundary of the ABPS—or the subsystem boundary if it is packaged together with the CSE.

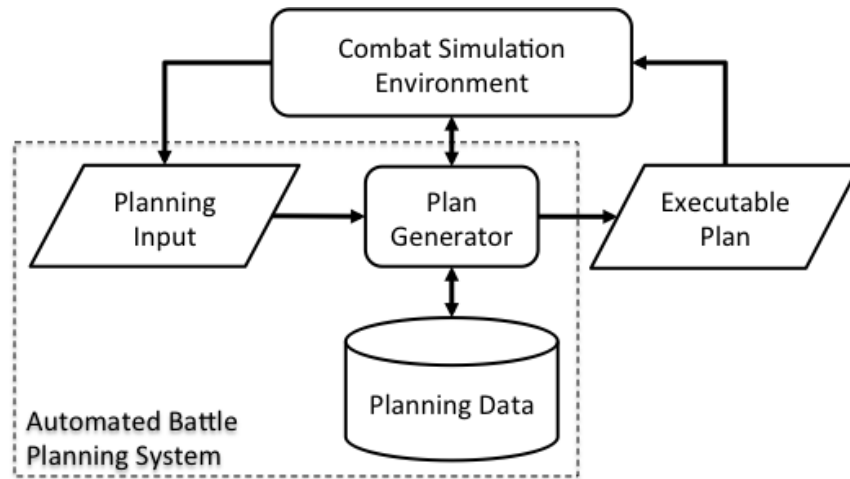


Figure 16. Top-level Framework

The Planning Input takes data from a scenario in the CSE, and from a user interface, and provides it to the Plan Generator. Critical information such as goals, objective functions, and unit configurations are contained in the Planning Input. The Plan Generator uses this input, its internal data structures, algorithms, and tools, along with the rules and heuristics of the Planning Data, to produce a plan in its own internal representation. Finally, it translates this plan into a format that can be processed by the CSE. This Executable Plan is provided to the CSE and used during one or more replications. The subcomponents and more specific interfaces of the framework are shown in the sections describing each component.

2. Other Useful Terms

In Chapter I, we introduced the terms *conceptual planner model* and *executable planner model*. Since we are now discussing the component parts of a planner, we

introduce the corresponding terms *conceptual subcomponent model* and *executable subcomponent model*. Subcomponent means that the interesting level of detail is one layer below the Planning Data, Planning Input, or Plan Generator shown in Figure 16. The *conceptual* subcomponent model is a theoretical construct used to describe concepts, data structures, and algorithms of a subcomponent. It may be specified with mathematical notation, pseudocode, UML, or descriptive text, to name a few examples. A conceptual subcomponent model is meant to handle the specific modeling decisions, level of detail, form of time advancement, combat adjudication method, etc., of its conceptual combat model. It takes advantage of the combat model's particular features and handles its specific challenges. The *executable* component model is a realization of a conceptual component model in an ABPS.

The framework is agnostic about when or how many times it is run. We categorize a few different options. *Offline planning* is defined as the generation of plans prior to CSE execution; this approach supports human review, acceptance, and possibly modification. *Online planning* means that plans are generated either immediately before execution or mid-execution with no opportunity for a human to reject them. A design where automatically-generated plans are used as opponent input for a human planner, such as in a tactical decision game, would still be considered online planning since the human planner does not decide authoritatively whether to reject the opponent's plan (short of quitting the computer program). Recall, from Chapter I, that we distinguish initial planning from replanning based on whether plans can be altered or replaced during a replication. Replanning is only compatible with online planning, but initial planning could be used for either online or offline planning. *Conditional planning* is defined as a special case of online planning where the ABPS is invoked as the result of conditional logic in the CSE, such as detecting when the current plan is ineffective. Planning conducted at set points in time or with a fixed periodicity is called *scheduled planning*.

The component descriptions below, unless otherwise specified, focus on a single invocation of the ABPS.

3. Separation of Duties

The Planning Data decouples military planning logic from more general AI-based algorithms found in the Plan Generator, in the same way that the developers of a planning software product (like a generic HTN planner) are separate from the teams that use the product to produce plans for a specific domain. This separation is meant to allow distinct configuration control on these two components, which can be managed by different groups with different skill sets.

In the video game development community, there is a distinction between level designers and AI developers. The former are responsible for crafting the gameplay experience using the tools provided by the latter. When a tool proves insufficient, designers call on developers to provide additional capability.

A combat model with an ABPS may actually need a three-way separation:

- *Scenario designers*, who choose terrain, military forces, objectives, locations, metrics, and are ultimately responsible for the tactical behavior of the modeled units (as analysts or training providers, for example). They configure the CSE and use the ABPS.
- *Behavior developers*, who encode tactical planning logic within the architecture of an ABPS. They modify the Planning Data but not the Plan Generator. Note that the term *behavior developer* is sometimes used to identify the specialists that write reactive tactical behaviors. We overload the term intentionally, since these same individuals probably have the right skills to work with the Planning Data.
- *Automated planning developers*, who are responsible for the architecture and algorithms of the ABPS. They modify the Plan Generator and the schema of the Planning Data, and they assist behavior developers with their responsibilities.

We need this additional silo because, unlike most video games, the scenario design and AI development for a combat model are often not done together on a single team. Instead, the combat model is distributed to a number of different user organizations, and each of those develops its own scenarios for specialized purposes. Sharing is encouraged, but different units have different missions, equipment, capabilities, and techniques, so there will always be some variety in the types of scenarios and plans needed at different sites. Scenario designers typically do not have the

programming background needed to encode tactical planning logic in a programming language-like format, so behavior developers with specialized skills are needed when the default Planning Data is insufficient.⁹ Finally, we distinguish automated planning developers from behavior developers because automated planning is an even more specialized skill than behavior programming, and a single Plan Generator is likely to be useful for a variety of different Planning Data configurations. Our support for this claim is that, as we mentioned above, various generic automated planners have proven useful when configured with tasks, methods, operators, actions, or chronicles—depending on the underlying approach—for different domains.

Video game AI developers, in contrast, can combine the behavior development with the automated planning development (for games that use automated planning) because they can work closely with the scenario designers and have knowledge, prior to release, of all scenarios and desired behaviors that will be shipped with the product. Behavior developers for a combat model are more like aftermarket scripting specialists for those few games and combat models that expose behavior scripting to the user community—for example, Bohemia Interactive’s ArmA series.¹⁰ If it turns out that two of these three silos can be combined for a particular ABPS or user group, no harm is done to the framework. The separation of Planning Data from the Plan Generator is still useful in terms of software and data organization.

4. Version Control

The framework advocates strict version control for all components. This is standard practice in software engineering, but we need to underline its importance for the combat modeling domain. If a user organization employs a set of scenarios involving automated planning, it may need to use those scenarios again in the future. For analysis, it may need to run new replications or expand upon a previous study from a known starting

⁹ Additional concepts and tools for integrating the efforts of tacticians and programmers are presented by Clark, Eichelberger, and Smith (2010). Their work provides further support for the separation of duties we discuss here.

¹⁰ Some user-written scripts are shared here:
http://www.armaholic.com/list.php?c=script_library_armas3

point. For training, it may receive a new training audience with similar needs to a previous one. If the ABPS changes between these current and future events, then the users may be unable to reproduce the same kinds of plans they did previously. Saving plans rather than recreating them is probably not sufficient because the saved plan does not contain the internal logic that the ABPS used to create it—updates to the saved plan could only be done manually, if at all. Even if the ABPS is somehow able to start from an existing plan and make small changes, it may be the tactics or character of planning that users would like to retain. This statement applies even if the ABPS has stochastic algorithms. If an ABPS is too unpredictable or unreliable under change, then users will steer away from using it. However, if “old scenarios” can run exactly like they did when originally tested (with respect to ABPS output), we can mitigate this concern. Users may upgrade scenarios to new ABPS versions, but they can fall back on the legacy capability in the worst case.

Each component is likely to require changes at different rates and for different reasons. The Planning Input only needs to change when a dependent part of the CSE changes or when a new feature of the ABPS requires an input that does not yet exist. The Plan Generator must change in response to any changes in its interfaces with the Planning Input, Planning Data, or Executable Plan. Developers may also choose to upgrade the internals of the Plan Generator; in some cases, this may affect only performance and not output, but even these types of changes should be tested. The Planning Data changes most often and more fluidly because it supports the needs of scenario designers, who work on different project cycles than the planning developers. We discuss special version control techniques for the Planning Data in its own section. For the other components, a single version control number for the entire ABPS should be sufficient. This could be subsumed in the CSE’s version control; if not, then each ABPS version should list the earliest supported CSE version.

B. PLANNING DATA

The framework includes a persistent data store called the *Planning Data* (Figure 17) in order to support a data-driven approach to the development of functionality. Some of the Planning Data is, in fact, executable code with a particular structure.

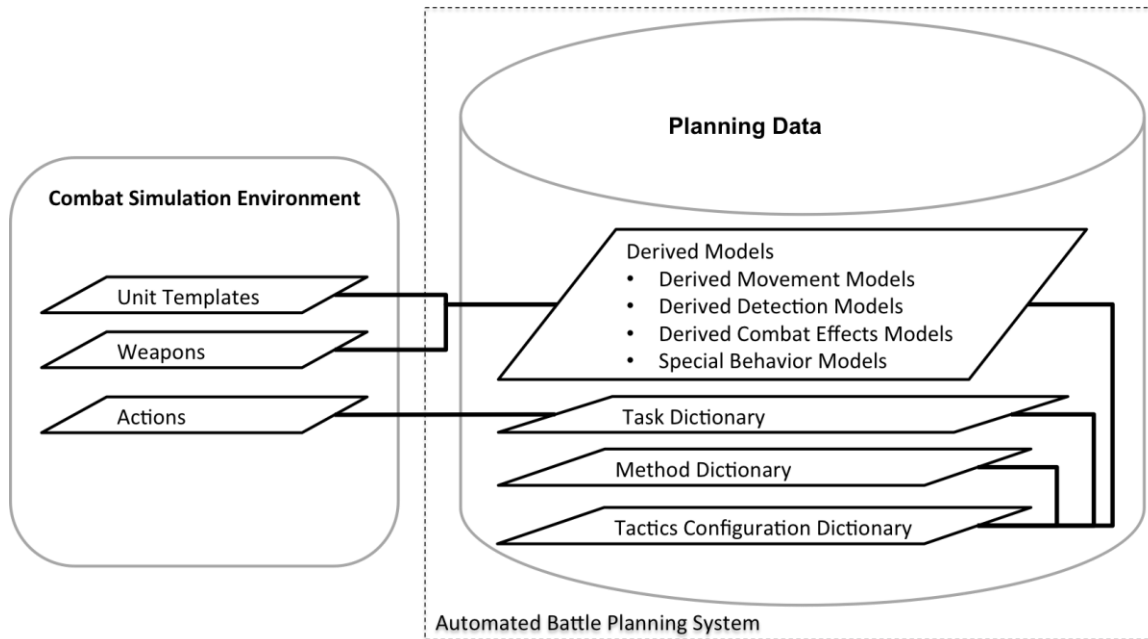


Figure 17. Planning Data

The Planning Data of this framework is meant to be an enterprise-managed database, not just a local user's data file. This supports the sharing of products between organizations and reduces overlapping effort. The *method set* concept, defined below, protects individual scenarios from unexpected changes in such a shared resource.

Each conceptual component of the Planning Data (the Task Dictionary, for example) may be realized with one or more separate data stores. Ideally, everything in the Planning Data adheres to a single, seamless data model. This would more easily support the sharing of information between components of the Plan Generator. This may not be possible, though, if some of its subcomponents were originally implemented for other domains. In this case, data format translation is necessary. We discuss this further in the Plan Generator section.

1. Definitions and Concepts

Let us establish a few definitions to help describe the Planning Data subcomponents.

a. *Units and Templates*

Units are just as we defined in Chapter I. We primarily work at the unit level, only occasionally referring to entities. To clarify, note that *unit* here refers to a software object that can be manipulated by the ABPS and corresponds to a particular unit instance in a scenario. It is an implementation decision whether this is a reference to a unit object in the CSE, a separate object linked by common name, etc. Each unit has an assigned *echelon*: one of a discrete set of values such as squad, platoon, company, or battalion. A unit's echelon does not determine its command relationships; it is simply a clue to the level of abstraction that the ABPS needs to use during planning. Small units that cannot be assigned tasks by the ABPS (see Commands and Hierarchies below) are not mentioned elsewhere in this framework. The term *unit* here refers to the objects on which the ABPS operates.

A *unit template* is a configuration of the attributes of a unit with the exception of explicit planning rules or planning guidance. If units contain entities, then each unit template also includes a multiset of *entity templates*. Similarly, an *entity template* is a configuration of the parameter values for an entity. The unit template includes reactive behavior models (which we abbreviate to *behaviors* henceforth), as discussed in Chapter I. Behaviors may be defined at the unit or entity level and may be built-in or configurable, depending on the CSE. The unit template does not include *planned* modification or toggling of those behaviors that are specific to a scenario.

Every unit is an instance of a unit template. Unit templates are distinguished from *unit profiles*, defined in more detail below, which contain planning guidance. If the CSE allows users to create units with unique attribute values that do not match a standard template, then we say, for terminology purposes, that each unique unit is the lone instance of a unique unit template. Two templates are considered equivalent if all of their parameter values (including the multiset of entity templates) are equivalent. Unit

templates may be named, and those with different names may still be equivalent but are always distinct.

A *partial unit template* is a unit template with some attributes left unspecified. Any unit template or unit whose attribute values are identical to the specified attributes of the partial unit template are said to *match* it. This is analogous to the concept of unification in first-order logic.

b. Commands and Hierarchies

The next few terms build to the concept of a command. This is really no different from the way we defined commands in Chapter I, but we gather some other useful terminology along the way. We begin with the *side*, a set of units that are aligned together under a common commander or within the same alliance, coalition, or unification. A *force* is a subset of a side. It follows that each side is a force.

Every unit is *commanded by* at most one other unit, called its *commanding unit*. The *commanded by* relationship is between individual units, not unit templates (even though some CSEs may include “templates” of commanded-by relationships). A *chain of command* for a unit d is defined recursively as the set of all units c such that

- d is commanded by c , or
- c is the commanding unit of a unit in the chain of command of d 's commanding unit

In other words, the chain of command is formed by following the path of *commanded by* relationships starting from d 's commanding unit. A *command* is a force in which the chain of command of every unit includes the same unit, called the *common commander*. If we interpret a command as a graph in which units are nodes and command relationships are edges, then a command is a tree.

There are a few different ways that combat models may interpret hierarchical unit organization. The planner must account for the approach in order for its output to be meaningful for the CSE.

- (1) *Contingent aggregation*: Each unit object represents the entire unit, including all subordinate units. Units may split into several subordinate

units (including, perhaps, a smaller version of the original unit such as a “company minus”) under certain conditions, such as the decision to maneuver separately or upon taking significant casualties. The opposite may be allowed as well: subordinate units that join into a single, larger unit. This might occur after an amphibious or airborne assault or after the movement phase of an infiltration. Note that simply hiding subordinate units for display purposes is not aggregation by our definition because it does not change the state of the executable combat model. For example, the MTWS “Aggregate” command (DOD 2014b, 143) hides subordinate units from the human user, but they can still be targeted by enemy units.

- (2) *Explicit command and control nodes*: Each unit object represents either an atomic unit (usually a small unit) or a command and control unit, such as a forward command post or a small unit command team. In this approach, the command and control units exist as targetable presences in the simulation. Commands are not represented with a single object, but with the entire hierarchy of small units rooted at the common commander.
- (3) *Abstract command and control nodes*: In this approach, only the atomic units have a presence in the executable simulation, but command and control nodes exist as abstract objects to provide guidance for how the atomic units should be employed in relation to each other. The ABPS may be designed to use this approach for a combat model with a “flat” organization—meaning that only one echelon is represented—to generate plans with the appearance of a higher-echelon organization.
- (4) Combinations and variations of approaches 1, 2, and 3 are possible. For example, atomic units under an explicit or abstract command hierarchy may disaggregate. Command and control nodes may be explicit only at certain echelons and implicit otherwise.

c. *Actions*

The *actions* of a CSE are the instructions that may be included in the Executable Plan. Actions are issued to units and entities, and they result in a change to the state of the CSE. Our actions are similar in concept to the actions of classical planning, but we do not assume a classical planning approach for the ABPS. Typical actions include movement to a waypoint, switching engagement modes (i.e., from “only when fired upon” to “engage upon target detection”), firing on a designated target area, and digging defensive positions. In our definition, actions are strictly part of the CSE and external to the ABPS (but see the Task Dictionary section below).

d. Derived Models

In general, an ABPS requires *derived models* to estimate the effects of many highly detailed actions. This allows planning to occur at a higher and less computationally demanding level of abstraction. The derived models are the specific functions, algorithms, equations, and other tools that make up the conceptual world of the ABPS, as described in the generic framework for planning in Chapter II. In this framework, the CSE is the external world; the real world is not even part of the framework. This can be a source of confusion, because the derived models are “models of the model” of the real world.

Derived models are essential for enabling an ABPS to efficiently reason about the future. Their development is a characteristic and rather unique aspect of automated battle planning, compared to other combat modeling skills. Here are some examples to help clarify how derived models work.

First, we have a movement example. Let us assume that some CSE has fine-grained equations to represent acceleration, effects of rough terrain on rate of movement, and realistic turning radii for each entity. It offers actions to instruct units to travel to a given waypoint, but the actual travel time depends on the frame-by-frame movement model, which is affected by the terrain type, number of turns, size and formation of the unit, and so on. Rather than use this detailed movement model to calculate paths or travel times for planning, we could develop a simpler, piecewise constant-velocity movement model with a small adjustment for unit size. If the simpler model can estimate the travel time within less than a second with high probability, say, then we have a derived movement model that is accurate enough for planning purposes and much less costly to calculate.

Our second example is a targeting model, in which each entity can only detect targets within a visibility cone extending in the direction its head or turret is facing, but the members of a unit scan their surrounding area reliably. The ABPS may not need to deal with directional facing at all. Instead, perhaps we could assume that each unit has enough entities to scan in nearly every direction. We could use an estimated detection

time counting from the moment a line segment between the detecting unit's and potential target unit's center points becomes unblocked. Developers could derive such an estimate through analysis of the target scanning code and entity-level detection model, by experimentation, or a combination of the two.

Derived models should be indexed by unit template so that units with equivalent templates can share the same models. If the CSE does not directly provide derived models—which is not common, since most do not currently have an ABPS—then behavior developers must create them. Furthermore, each new or changed unit template (to include each new uniquely-configured unit) requires new or updated derived models. Therefore, parameterized or self-configuring derived models are desirable. A simple example of this is a derived model of movement time that uses the slowest entity in the unit for each portion of the path. This would provide a worst-case estimate for a unit with heterogeneous movement capabilities.

Verification is an important requirement for derived models. Note, however, that estimates are usually sufficient. More precision is better if the computational budget is available, but even the most precise calculation is only an estimate when applied to a stochastic combat model—or even a complex deterministic model run with a variety of starting conditions. This might sound like shaky ground on which to build plans, but consider that human scenario designers who author battle plans manually must do the same thing. Real world planners are faced with an even greater level of uncertainty, since they must account for the unknowable future decisions of the other human actors on the battlefield and the greater opportunities for unexpected events in the real world.

e. Distributed Version Control

Since the Planning Data is a shared resource used by multiple scenario design teams and updated by multiple behavior developers, there is a tension between version control and sharing of work. Let us assume for a moment that the ABPS itself is a fixed version: the Planning Input, Plan Generator, and even the schema of the Planning Data do not change. Nevertheless, the contents of the Planning Data—the tasks in the Task Dictionary, methods in the Method Dictionary, models in the Movement Capabilities, and

so on—are still updated constantly as behavior developers create the tools needed by scenario designers for their unique requirements. A scenario designer may be happy to find some useful tools in the Planning Data that were created by some other group, but it would be disastrous to have such a tool change unexpectedly in the middle of a study or training event.

The extreme approach to preventing this issue is to separate the Planning Data into enterprise and local stores, changing the global store only with ABPS version updates. New or updated entries in the Planning Data could be submitted to a centralized change control board and tested prior to addition. However, this would permit sharing of work only during version updates. Each scenario designer would be unlikely to risk an upgrade during a project, further delaying the sharing of work. Furthermore, sharing is much less likely with this “push” approach, since it puts an obstacle (the change control process) in the path of behavior developers who have already fulfilled the need of their local scenario designers. Under this scheme, local Planning Data is likely to remain local.

However, an update-only approach may strike a better balance. In this scheme, each change to a task, method, derived model, or configuration entry of the Planning Data is posted as a separate, new version. All previous versions are retained, so older scenarios that depend on specific planning tactics, models, or heuristics may continue to invoke the same versions. Scenario designers may choose to upgrade at any time, of course. This scheme can be achieved with a post-and-download approach (users only download new versions when ready to try them), which is exactly what a developer employing multiple open source resources would do. Indeed, a branching and forking system like that of Git¹¹ is a useful way to organize and display multiple versions. But a “pull” approach, in which new entries and versions are always stored in a single repository rather than pushed when a developer feels like dealing with the hurdles, better promotes sharing. In place of a change control board barring entry, the commonplace five-star user review system can provide a useful indication of quality. An enterprise review process could provide a stamp of approval (or caution), as able, to supplement the

¹¹ <https://git-scm.com/>

user reviews. Of course, storing a new version with every hourly “file/save” is not realistic. A good time to publish new versions is when one person provides a tool to another—in this case, when a behavior developer provides a new version to a scenario designer.

Ultimately, the approach to organizing and managing the Planning Data is an architectural decision. The important point is that automated battle planning requires multiple individuals with different skills, working in separate organizations and locations. It would be a mistake to view it as a single-user desktop application.

We now cover the subcomponents of the Planning Data.

2. *Derived Model Dictionary*

All of the derived models are stored together—conceptually, if not physically—in a store called the Derived Model Dictionary. We describe a few categories of derived models that are likely to be present, followed by an additional placeholder, the Special Behavior Rules, for derived models that do not clearly fit into one of the standard categories. The intent is to extend the framework with additional categories as they coalesce.

a. Derived Movement Models

If the battle planning system is to generate a plan with movement, it probably needs derived models for the movement of its units, called *derived movement models*. These are separate from any movement-related tasks (see the Task Dictionary section below) and are used as inputs to the Pathfinding component (described in the Plan Generator section). The linear movement estimate of the previous section is one example of a movement model.

At minimum, an ABPS must be able to determine whether some part of a map is traversable by a given unit, and at what speed. If the CSE includes combat formations, then the set of allowed formations and combat performance parameters for each should be available in the Planning Data. For each formation, the plan generator is likely to need

the time to change from one formation to another, the shape and area covered by the formation, and the effective rate of movement of the unit while in that formation.

b. Derived Detection Models

The detailed target detection models of a CSE—that is, the detection calculations used during replications—are accessible through unit templates. *Derived detection models*, which allow the ABPS to reason about target detection tractably at the unit level, are likely necessary in order to generate effective, executable plans. However, some unit-level combat models may have detection algorithms simple enough to be employed as-is. Derived detection models should be indexed by ordered pairs of unit templates (i.e., one type of unit attempting to detect another type of unit) and parameterized by information such as distance, movement status, and intervening terrain. They may provide information such as

- (1) Detection field or detection cone shape and size
- (2) Expected time to detect
- (3) Rules for simultaneous targets: priority, multiple detections, etc.

c. Derived Combat Effects Models

CSEs, in general, have simulated weapons. These may reside within the unit objects or be references to separate weapon classes. The Plan Generator needs *derived combat effects models* indexed by unit template pairs. These model the cumulative results of weapon employment over time, such as attrition and suppression. The full combat adjudication algorithms and attribute values of each weapon are less useful for forward planning because they are written for instantaneous, repetitive calculation. The ABPS may require combat effects information such as

- (1) Area, volume, or set of navigation nodes threatened by each unit
- (2) Attrition effects (killing rate) given parameters such as target range, terrain protection, combat formations, and direction to target relative to the combat formation
- (3) Suppression effects given the same kinds of parameters listed for attrition effects (Chapter IV describes suppression effects in more detail)

(4) Estimated firing time based on ammunition load-out

One possible approach to a derived combat effects model is an *aggregate unit weapon* that combines the capabilities of all weapons in the unit. For a unit with heterogeneous armament—which is almost any type of unit in western militaries—effects may need to be defined piecewise to deal with differing maximum ranges. As always, the level of precision depends on the requirements for the ABPS.

Some examples of derived combat effects models are the combat outcome prediction techniques used in automated RTS game-playing research (Churchill and Buro 2013; Stanescu, Barriga, and Buro 2015; Uriarte and Ontañón 2015). Where possible, indexing derived combat effects models by weapon type (rather than by unit template) may be more efficient. The number of different weapon models is often less than the number of unit templates.

d. *Special Behavior Models*

Special behavior models is a catch-all category of derived models that do not fit well in the derived movement models, derived detection models, or derived combat effects models. These might be needed for support units or non-combat units (i.e., civilians) with properties not well represented by measures of location, time, or fighting strength, but that still affect the expected outcome of the plan. For example, imagine a scenario in which a command is conducting an attack near a civilian population center. The scenario includes a model of civilian traffic patterns. We might need a special behavior model to estimate the risk of civilian traffic fouling the attack plan based on the planned maneuvers of combat units. This would allow us to incorporate protection of civilians into the goal test or objective function of the Plan Generator.

Special behavior models could cover almost anything, but their use is the exception rather than the rule. They are only needed if a particular behavior is known to have a measurable and at least somewhat predictable effect on the results of plans. Before creating a special behavior model, developers should first consider whether derived movement models, derived detection models, or derived combat effects models could sufficiently represent the net effects of the behavior of interest.

3. Task Dictionary

The Task Dictionary stores the set of *tasks* that can be assigned to units. We borrow the term *task* from HTN planning, but the framework does not require a task network representation or an HTN planning algorithm. Some of the tasks map to the CSE's actions, but the planner may augment the task language with other tasks (see *operational tasks* below). The planning system may exclude from its Task Dictionary any CSE actions that are not useful for its planning purposes.

Each task may include preconditions, parameters, and ranges of allowed parameter values. Preconditions are defined as they are in classical planning: each describes a partial configuration of the conceptual world under which the task is allowed to be initiated. The tasks do not, by themselves, provide any guidance about when they *should* be used (see Methods), only when they *may* be used (preconditions). Parameters (within the allowed value ranges) allow behavior developers to write tasks that are applicable for a variety of different situations. They tend to reduce the number of different tasks needed at the price of greater complexity per task. Tasks may be *instantaneous* (a discrete value change at a single point in simulation time) or *enduring* (affecting the unit state in a certain way between initiation and termination). Although these two types are analogous to the events and persistence conditions of continuous-time planning, the framework does not stipulate a particular time representation.

We distinguish *tasks* from *task instances* in the same way that HTN planning does (but, again, we do not require an HTN approach). A *task* is an entry in the Task Dictionary. A *task instance* is a copy of a task assigned to a unit in the conceptual world of the ABPS. Task instances may have partially or fully bound parameter values during planning.

Each task is classified as an *operational task* or an *operational action*. Operational tasks are assigned to command and control units that have subordinates. They do not directly cause any change to the state of the CSE. Instead, their purpose is to provide a higher-echelon description of the combination of tasks being performed by the subordinate commands. Operational tasks are similar to and inspired by the nonprimitive

tasks of HTN planning, with the distinction that they must be assigned to units that have subordinates in a command hierarchy. Since they have no direct effect on the CSE, they may be assigned within the ABPS to both explicit and abstract command and control units.

Operational actions are much like the operators of classical planning and HTNs. Each operational action corresponds to one or more action in the CSE. By definition, an operational action with fully bound parameters can be directly assembled into one or more CSE action without any need for further planning. Note that both operational actions and CSE actions have parameters, but CSE actions cannot be partially bound—they are run-time instructions that cannot execute without all required arguments (default values are possible, but still constitute bound variables).

Explicit command and control nodes, if part of the CSE, may be assigned both operational tasks and operational actions. Their operational tasks describe the high-level instructions for their subordinate commands, and their operational actions are for the fine-grained activity of the headquarters unit itself. Possible operational actions for a command and control node are moving to a new command post location and setting up communications equipment.

4. Method Dictionary

A method (we borrow the term from HTN planning) is defined as an optional subroutine, function, or mapping that generates planning branches. Of course, the HTN method is one possible format; control rules (Ghallab, Nau, and Traverso 217–226) and custom heuristics are others. Different types of methods may be present in the Method Dictionary if the Plan Generator uses multiple planning representations and algorithms. The Method Dictionary conceptually groups all methods together to help explain other concepts such as method sets (defined in the Input section). Classical planning approaches may use no methods at all (other than trying all noncyclic possible combinations), but some implementations (such as GOAP [Orkin 2005]) use search heuristics that might be exposed as configurable objects.

Each entry in the Method Dictionary may include a set of partial unit templates called its *applicable template set*. To apply the rule to a unit, the unit must match a member of the applicable template set. The inverse of this mapping is the *applicable method set*. The applicable method set of a unit template is the set of all methods for which it matches a member of the applicable template set.

The development of methods could benefit from collaboration between tacticians and behavior developers using an approach such as the Tesla language (Clark, Eichelberger, and Smith 2010). However, the Method Dictionary contains the executable results of a tactics development process, not a graphical front-end like the Tesla Editor Application.

5. Standard Tactics Configuration Dictionary

An enterprise Planning Data store could grow quite large and include a wide variety of tactics, many of which would be inappropriate or undesired for the units in a particular scenario. Tactics that would only be used by certain nations or units facing special conditions should only be available where those conditions apply. *Standard tactics configurations* group Planning Data entries together into coherent toolsets. To explain how they work, we first define the *planning style* and the *tactics configuration*, which are local to a scenario. The Standard Tactics Configuration Dictionary is a collection of standard tactics configurations.

a. Planning Styles

Intuitively, a planning style restricts individual units to the planning assumptions and tactics appropriate for their training, experience, motivation, goals, or national military culture, according to the scenario designer. The planning style concept is inspired by the style concept of the behavior tree pattern (Isla 2005), which we discussed in Chapter II. Behavior tree styles allows designers to restrict entities to a subset of their script nodes. Similarly, planning styles allow designers to restrict the ABPS to a subset of the entries in its Planning Data—tasks, methods, and derived models.

A *method set* is defined as a subset of an applicable method set (which is presented above in the Method Dictionary section) with version information and optional binding constraints on its parameters. The version information ensures that each invocation of a method in the method set retrieves the same version of the same method. The binding constraints, when used, provide a more restrictive range of parameter values than the general binding constraints of each method. *Task sets* and *derived model sets* are defined similarly: subsets of the Task Dictionary and Derived Model Dictionary, respectively, with version information and optional binding constraints.

A planning style is the combination of a method set, a task set, and a derived model set. The relative importance of these three elements depends on the internals of the Plan Generator—for example, an HTN planning implementation may be sufficiently specified by a method set, while a classical planning implementation (which would have no methods) would require a task set. Two planning styles are equivalent if their method sets, task sets, and derived model sets are equivalent. Each unit may be assigned one planning style.

If two units with equivalent unit templates are assigned two different planning styles, their roles in the generated plan may end up being very different. In this sense, the planning style alone is enough to distinguish a “type” of unit. We introduce a new term to highlight this: a *unit profile* is a unit template together with a planning style. Two units have equivalent unit profiles if and only if they match the same unit template and have equivalent planning styles assigned. A unit with a unique unit template necessarily has a unique unit profile.

The Plan Generator is required to ignore all planning branches for a unit that do not match an entry in that unit’s planning style. Similarly, it must use the derived models provided in the planning style, not newer versions that may reside in the Derived Model Dictionary.

b. Standard Tactics Configurations

A *tactics configuration* is a mapping from units to planning styles. Every scenario that employs the ABPS must have a tactics configuration. A *standard tactics*

configuration is a mapping from unit *templates* to method sets—in other words, it specifies a set of unit profiles. Standard tactics configurations are shortcuts for quickly generating or resetting the tactics configuration of a scenario: every unit in the scenario is assigned the planning style listed for its unit template. A scenario designer can avoid dealing with tactics configurations by using a standard tactics configuration that has been verified and, ideally, validated for the class of scenario in use. However, the (nonstandard) tactics configuration provides additional configurability when needed.

Each unit template in a standard tactics configuration maps to only one planning style. If different planning styles for a particular unit template are desired, behavior developers can create copies of the unit template and give each copy a unique name. The standard tactics configuration can map the distinctly named (but equivalent) unit templates to different planning styles. Even if the CSE is ignorant of the planning system, this approach still lets us invoke the appropriate unit profile from the CSE input by selecting the correctly named unit template.

C. PLANNING INPUT

Most of the input to the planner is drawn from the CSE. We think of the CSE as being devoid of a planning system. If it has one, we place that part of the environment within the boundary of the ABPS. The Planning Input component (Figure 18) is not chiefly a processing component, but user interfaces could be found here. Concrete settings for all of the required elements of the Planning Input constitute a single instance of an automated battle planning problem.

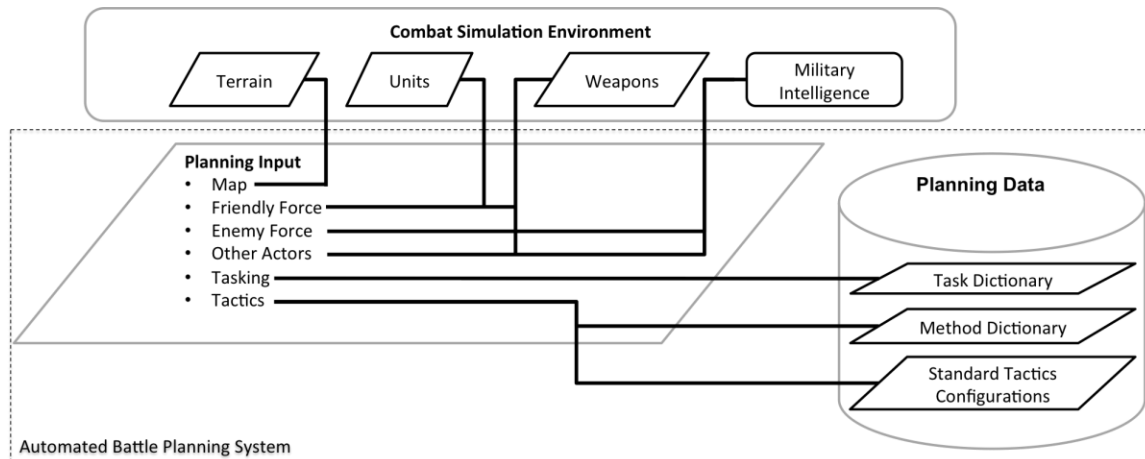


Figure 18. Planning Input

1. Map

The map is a digital representation of the geographical terrain where the combat occurs. The details included in the map depend on the terrain representation of the combat model, but the important point here is that the planner can access those details to support the prediction of combat outcomes. The planning framework is agnostic about whether it has direct access to the simulation environment's terrain objects; a copy or sufficiently accurate surrogate is acceptable (see the Annotated Mobility Graph under the Planner section below).

Typical features and functions of a Map are described in the remainder of this section. We list the elements that are most likely needed to construct the Annotated Mobility Graph.

a. *Coordinate System*

The map can be indexed by coordinates. The planner can access detailed information at any given coordinates within the bounds of the map.

b. *Ground Elevation*

The ground surface is described by a terrain elevation model. Given horizontal coordinates, we can look up the corresponding terrain elevation.

c. Obstacles

Objects that affect detection, targeting, or the mobility of units must be accessible (i.e., part of the Map). The obstacles are assumed to be defined by sets of polyhedrons. The ground elevation data and set of obstacles together describe the areas that may be occupied by ground units.

Buildings, fortifications, trees, and any other object that may have an impact on combat results are considered obstacles. Platform-type obstacles (usually buildings) may create an environment in which units may overlap horizontally (in the xz plane) yet be at different elevations (have different y -coordinates), such as when different units occupy different floors of a multi-story building.

Some CSEs use volumes above the ground elevation to represent vegetation, debris, or some other wide area obstruction field. These typically affect detection, targeting, or mobility just as we have described obstacles, so we include them in the obstacle category. Some obstacles are impassable to units, while others only modify movement parameters (usually slowing the units down).

2. Friendly Force

The Friendly Force is the set of combat units for which the ABPS will generate a plan. All units in the Friendly Force are assumed to be on the same side, but it is possible to run the planner with a Friendly Force that does not constitute an entire side. If that is the case, all other units on the same side are considered Other Actors (see below).

If a single planning system is to be used to generate plans for more than one side of the engagement (such as the attacker and the defender), then it must be run as two separate instances with two different Friendly Forces. The planning system does not directly provide information from one instance to another (but see the Enemy Force section below).

We would expect Friendly Force information to be drawn from the CSE's Units system and (if separate) its Weapons system. Some of this information might reside in

unit templates, but at least a portion of it must be scenario-specific, such as unit starting locations. The planner assumes that the Friendly Force information is accurate.

The following information is expected to be available as part of the Friendly Force input. Note that particular subcomponents of the planner do not necessarily require all of these inputs.

a. Unit Organization

Any hierarchical command relationships between the Friendly Force units must be supplied as input. The planning system may employ a *single-command assumption*, meaning that all Friendly Force units are assumed to fall under the same command. Under this assumption, any Friendly Force units with no commanded-by relationship are assumed to be sibling units commanded by a single unnamed common commander. Without the single-command assumption, the planning system may need to separately plan for the different commands in its input, which could be handled in the Input Processor or Planning Controller (described in the Plan Generator section).

If support relationships are needed for planning, then they must be provided as part of the unit organization. In accordance with the doctrinal definition of support relationships (described in Appendix C), each support relationship should include a type, category, supported command, and supporting command. Examples of types are fire support and logistical support; examples of categories are direct support and general support. Although a unit may be commanded by only one other unit, it may be involved in several different support relationships, supported in some and supporting in others. Support relations may be defined on the union of abstract and explicit units in order to support guidance for planning. A support relationship from an abstract command and control unit may not function as intended during run-time if the CSE cannot handle immediate support requests (such as on-call fire missions for a direct support artillery unit), but this does not necessarily prevent the planner from doing its job of creating an appropriate initial plan with the information at hand.

Additional command and support relationships not included in the CSE may be allowed as input to the planner—i.e., as direct user input. This could be done with either

explicit or abstract command hierarchies. This design would cause the planner to generate plans that are structured and guided by an “unseen” hierarchy, from the perspective of the CSE. Under the right conditions, this may result in a richer and more meaningful simulation scenario by imposing some military organization where none is actually defined in the CSE.

b. Unit Starting Locations

The initial location of each Friendly Force unit, within the area of operations (see below), must be provided. In the special case of defensive operations with unlimited preparation time, this requirement is waived and the Friendly Forces may be positioned by the planner anywhere in the area of operations.

3. Enemy Force

The Enemy Force input has all of the same parameters as the Friendly Force input, so we do not repeat them here. However, in general the planning system cannot be assumed to have accurate information about the enemy. For this reason, we show Enemy Force data connected to a Military Intelligence module in the CSE. The Conceptual Planning Framework does not include the intelligence analysis process, which proceeds from intelligence requirements to data collection and processing, and eventually results in the dissemination of estimates about the Enemy Force (DOD 2009b). Instead, the ABPS is the recipient of the final stage of this process.

a. Multiple Values

The framework allows an optional confidence level, scored between 0 and 1 (inclusive), for Enemy Force parameters. If the CSE has no Military Intelligence module, we may proceed by providing Enemy Unit ground truth from the CSE’s Units and fixing the confidence level at 1. Confidence levels can be used to support decisions such as what type of operations are appropriate--a deliberate attack versus a movement to contact, for example (we would not plan a deliberate attack when Enemy Force positions have very low confidence).

If the Military Intelligence component provides a finite *set* of possible parameter values rather than single values, then the Enemy Force input parameters can be structured also as sets of values, each with a corresponding confidence level. If all n values are equally likely for a certain parameter, then they can each be set to a confidence level of $\frac{1}{n}$. This is most likely applicable for Enemy Force starting locations. The following parameters are the ones most likely to use value sets:

- Unit Organization
- Starting Locations
- Weapon Types and Counts

Another possible approach to multiple parameter values is to describe some parameters as random variables with a simple probability distribution. For example, a unit location might be described as a bivariate normal distribution on its x and z coordinates. The Plan Generator might approach this by using an expected value or the upper and lower bounds of a confidence interval; these may be provided with the input or generated by the planner if given an expression for the distribution. This mathematically detailed approach to Military Intelligence is not common in real world planning, but might be used in the context of analysis.

b. Plan

For Friendly Force defensive operations in particular, the assumed plan of the Enemy Force is a significant input to the planner. Of course, we do not have this input for the Friendly Force because the whole purpose of the planner is to generate the plan.¹² In fact, the Military Intelligence component of the CSE could use a separate instance of the same type of planner to generate Enemy Force plans as input to the Friendly Force planner. Whether or not it does so, Enemy Force plan generation is a Military Intelligence function from the perspective of the Friendly Force.

¹² In the case of replanning, the Prior Plan is part of the Tasking, but the Friendly Force still does not have an *updated* plan until the ABPS generates one.

In course of action wargaming (DOD 2011b), separate teams (of humans) simulate the execution of two competing COAs (embryonic plans for the Friendly and Enemy Force), making decisions that effectively change them or specify them in greater detail. Whether the ABPS may manipulate the Enemy Force Plan input in this sort of way during the planning effort is an implementation decision.

c. Multiple Scenarios

Standard U.S. doctrine calls for two Enemy Force COAs as planning input: Most Likely and Most Dangerous (DOD 2009b). If the Military Intelligence module provides two (or more) scenarios in this way, then the Planning Input may include multiple scenario files—one per Enemy Force COA. However, the ABPS is still expected to output a single plan, for a single target scenario, that deals with all input plans. To handle the multiple Enemy Force options, the ABPS developers must decide whether to complicate the Executable Plan output with branches and sequels or postpone branch planning to a replanning event (if online planning is an option).

4. Other Actors

Other Actors are any decision-making agents not contained in the Friendly Force or Enemy Force. As with the Enemy Force, the Friendly Force cannot be expected to have intrinsic knowledge of Other Actors' plans or other parameters, so we say that their information comes from the Military Intelligence component of the CSE. If that component does not exist, we can provide either ground truth through the CSE's Units system or from manual input, which would assume a "perfect" Military Intelligence model.

Other Actors could include neutral combat units, local citizens, international aid workers, or even herds of livestock. In some cases, it may be appropriate to include a plan for some of the Other Actors in the same way that a plan can be part of the Enemy Force input. For example, the Friendly Force might be aware of an international aid convoy's plan to provide food to some local villages. Other Actors may have reactive behaviors, which may require derived models; these would be stored in the Planning Data.

The Framework can be made applicable to Stability Operations, at least in theory, by eliminating the Enemy Force input and including Other Actors for the different factions and organizations involved in the scenario. However, the time scale of Stability Operations and the specificity of plans for their conduct are usually very different from what we expect in conventional combat operations. In some stability scenarios, groups of Other Actors may become violent against Friendly Forces or a protected group. If so, and the modeled rules of engagement permit, the violent actors may be re-interpreted temporarily as Enemy Forces and dealt with using conventional battle planning. Unless the violent events are highly predictable, this would require an online planning capability. The responsibility for reclassifying Other Actors to Enemy Forces, or vice versa, falls to Military Intelligence—not the planning system.

For a particular planning instance, Other Actors includes the higher headquarters and adjacent units that are on the same side as the Friendly Force but not subject to the planner's output. Doctrinally, the Friendly Force should have the plan (operations order) of its higher headquarters before it begins its own planning, and its Tasking (see below) should be drawn from the tasks contained in the higher headquarters plan. We revisit this point when describing the Plan Generator.

5. Tasking

The Planning Input must include a Tasking, which provides requirements and guidance to the planner. The Tasking could come directly from a user or from a software system (such as another instance of the ABPS).

We use the term *Tasking* rather than *Mission* for a specific reason. A doctrinal mission statement (i.e., for humans) contains only essential tasks and includes the *who*, *what*, *when*, *where*, and *why* of the required military action (DOD 2011b). In doctrinal planning, the mission statement is developed by the commander and staff of the planning unit, not by the higher echelon command tasking the planning unit. If we think of the planning system as a simulation of the commander's and staff's planning activity, then the responsibility of developing a mission statement lies within the system boundary, not as an input. The Planning Input provider plays the role of the higher echelon tasking

command, which has the responsibility of providing taskings to its subordinates in its own plan.

In all likelihood, the planning system does not actually generate a doctrinal mission statement. A software system does get the same benefit from such a “succinct” description of its most important tasks that a human planning team would. A likely design approach is to simply view all input taskings as essential and prioritize effort according to an objective function.

The Tasking consists of the following information.

a. Required Tasks

Required Tasks are task instances that describe what must be done by the Friendly Force. Required Tasks must be drawn from the *input tasks*, which are defined as a subset of the Task Dictionary along with ranges of allowed parameter values (which may be no less restrictive than the general value ranges in the Task Dictionary). The input tasks are presented to the user or exposed in an API; they are effectively a catalog of the ABPS’s planning capabilities.

A choice of Required Tasks for a planning instance comprises the *what* of a tasking. The *who* of the tasking is, of course, the Friendly Force, but the planning system may apply just a portion of its units against each Required Task.

Tactical task (DOD 2016a) is a doctrinal military term, not a term of this framework. However, a ground combat tactician would expect an ABPS for ground units to include at least some of the doctrinal tactical tasks in its catalog of input tasks. Here is a non-exhaustive list of tasks to consider, with definitions tailored to the terminology of this framework.

- (1) Seize: occupy a set of locations or areas on the Map, eliminating Enemy Force resistance as necessary.
- (2) Destroy: render the Enemy Force combat ineffective by eliminating some percentage of its units or entities
- (3) Block: prevent the Enemy Force from passing a specified point or line on the Map

- (4) Fix: prevent the Enemy Force from moving outside a designated area on the map
- (5) Retain: prevent the Enemy Force from occupying a designated area of the Map

Common parameters for tactical tasks, and which of the “five Ws” questions they help answer, are

- (1) Objective points, lines, or areas on the Map (“where”)
- (2) Direction of attack, specified as a range of horizontal directions (ground unit tasks use a two-dimensional, “overhead” perspective) allowed for movement and firing with respect to some objective points (“where”)
- (3) Percentage of Enemy Force to be eliminated (“what”)
- (4) Time interval for the task (“when”)

b. Preferred Tasks

Preferred Tasks are identical to Required Tasks, except that failure to achieve any or all of them does not constitute a failed planning effort. The ABPS should attempt to achieve as many of the Preferred Tasks as possible in the time available (if time-limited), but should prioritize Preferred Tasks after all Required Tasks. Developers may offer a more refined prioritization scheme for Preferred Tasks.

c. Prior Plan

As an optional feature for a replanning implementation, the Planning Input may accept a partially completed Executable Plan called a Prior Plan. If so, the Plan Generator can serve as a plan repair system. The Prior Plan must have annotations indicating which tasks are completed, failed, in progress, or have yet to begin. Even tasks scheduled for future execution may be marked as failed—for example, the CSE may have determined that a unit is unable to reach a start point at a scheduled time due to an unexpected enemy blocking position.

The Required Tasks may be partially or entirely contained inside the Prior Plan, so measures must be in place to avoid planning against duplicate tasks. Any tasks in the Prior Plan that are not Required Tasks may be changed or replaced during plan repair,

and all failed tasks must be removed. Generally, plan repair should change the Prior Plan as little as possible, but starting over from scratch is not out of the question. It would be the only option if the CSE could not provide an appropriately annotated Prior Plan or the ABPS could not accept one.

Replanning is simpler when the Executable Plan and Partial Plan (described in the Plan Generator section) share the same format. Otherwise, the Input Processor (also part of the Plan Generator) must do the reverse of the Plan Compiler. Since operational tasks may not be retained in the Executable Plan (by definition, they have no effect on the CSE), the Plan Generator may need to retain its last-generated Partial Plan to simplify “decompilation” of the Executable Plan.

d. Area of Operations

The Area of Operations (AO) restricts the Friendly Force’s movement and effects of fires to a region of the Map.¹³ All tasks in the Executable Plan must be constrained within the AO. According to current doctrine, friendly forces may move and fire across AO boundaries after coordinating with the owner of the violated AO. This can be handled for some scenarios by allowing AOs that overlap each other and by specifying the AO for movement and the AO for effects of fires separately.

e. User Cost Function

The User Cost Function is the user-configurable part of the Plan Generator’s objective functions. It allows the user to provide more nuanced guidance to the ABPS by weighting measurable features of candidate plans. For example, the User Cost Function could be a linear combination of several primitive cost functions. By modifying the comparative weighting of the scoring functions, the user should be able to cause the planner to generate plans that strike a balance of several important features. An example of this kind of multi-function balance is the *commander’s guidance* in the BBE system (Schlabach 2010), which we outlined in Chapter II. In current doctrinal terms, multi-function scoring resembles the *commander’s evaluation criteria* (DOD 2011b). An

¹³ Non-contiguous AOs are doctrinally allowable, but usually only for units with air assets.

important distinction between real world and automated planning is that a real world commander is not obligated to use the plan with the highest quantitative score; many subjective factors learned through experience, training, and education can come into play. An ABPS is probably limited to ranking options with a numerical score, but the framework does not stipulate that the User Cost Function be the only measure. Additional scores and conditional logic may be “hard-coded” into the Plan Generator to help control the realism, simplicity, computational demand, or even randomness (if desired) of generated plans. These and heuristic cost functions for guiding plan search are generally not user-configurable due to the development skills and testing needed to get them working correctly.

For ease of use, the ABPS should provide one or more well-tested defaults settings, with descriptive names, for the User Cost Function. Having comparable ranges of weights for different primitive scoring functions (i.e., all of the same order of magnitude) can reduce the trial-and-error burden on scenario designers.

Some potential primitive scoring functions that a planner could offer are

- (1) Time to complete all tactical tasks
- (2) Total distance covered by all Friendly Force units
- (3) Expected fraction of Friendly Force that will be lost
- (4) Expected fraction of Enemy Force that will be lost
- (5) Expected fractional exchange ratio (Helmbold and Kahn 1986)
- (6) Expected ammunition expenditure
- (7) Expected fuel expenditure
- (8) Fraction of objectives expected to be cleared of Enemy Force units

Not all of these scoring functions have the same properties with respect to plan search. For example, it may be difficult to find a good plan by following a search path with nondecreasing expected Friendly Force losses, as demonstrated in the next chapter. On the other hand, if there are no fueling sources in the scenario, then fuel expenditure is nondecreasing for each addition of a task. Implementation choices for the Plan Generator

may disallow some combinations of primitive cost functions. Planning developers must consider degenerate cases before offering configurable cost functions to scenario designers.

6. Tactics

In western-style military operations, a tasking provides maximum flexibility to the tasked commander by specifying *what* to accomplish (the tactical task) but not *how*. For M&S, we may require more control. For example, the purpose of a study might be to compare the casualty rates of a penetration versus an infiltration in a given scenario. A training application might need to pit the trainees as operational planners against the doctrinal style of several different threat nations. By allowing the user to constrain or modify the tactics of the planner, we provide more control and flexibility (to the user).

If unique tactical personalities are desired for simulated commanders, then certain units could be assigned unique planning styles. One-off planning styles are not stored in the Tactics Configuration Dictionary (unless the user expects them to be useful for future work), but with the scenario files.

a. Tactics Configuration

The Input may include a tactics configuration. This may be

- A standard tactics configuration drawn from the Tactics Configuration Dictionary
- A unique tactics configuration designed for the scenario
- A standard tactics configuration augmented with additional planning styles for the unit templates that do not appear in the standard tactics configuration

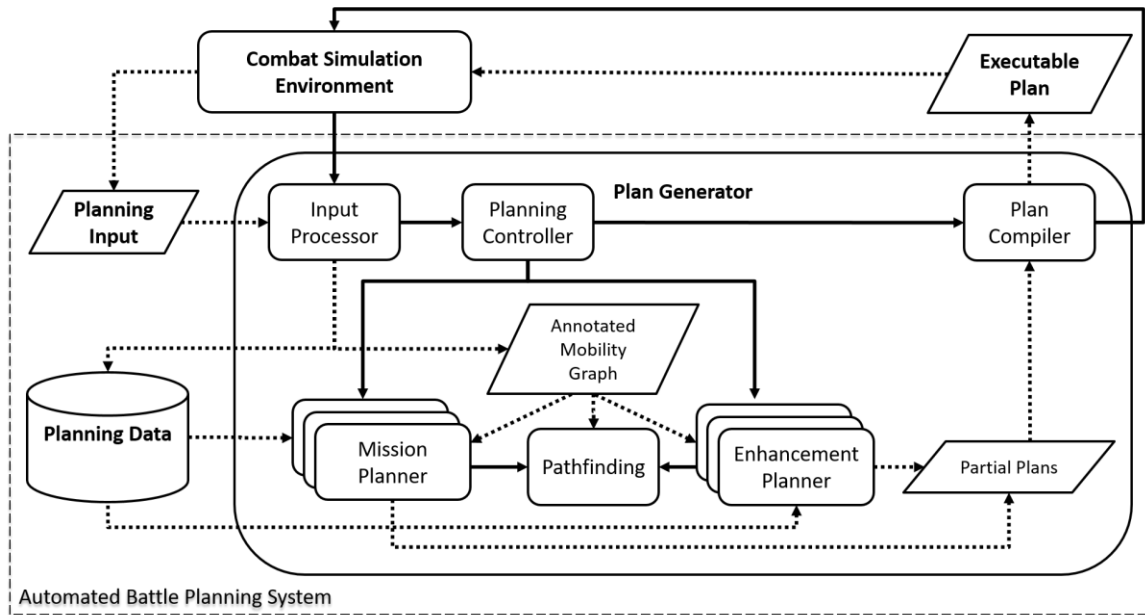
b. Forms of Maneuver

Scenario designers often have military backgrounds, so it is important to offer tactics configurations in the terms of the domain. For ground maneuver planning, these are embodied in the forms of maneuver (see Section J of Appendix C). More specifically, the user might want to restrict the form of maneuver of a designated unit (probably the

highest-echelon unit of the Friendly Force). Restrictive standard tactics configurations could be developed to support this level of control.

D. PLAN GENERATOR

The Plan Generator (Figure 19) consists of processing components that use the Planning Input and the Planning Data to produce an executable plan.



Solid lines represent control flow, and dotted lines represent data flow.

Figure 19. The Plan Generator

1. Input Processor

The Input Processor is invoked by the CSE or by the user to initiate execution of the ABPS. It retrieves necessary information from the Planning Data into local memory, generates the Annotated Mobility Graph (if it does not already exist), and passes control to the Planning Controller. If the Planning Input includes a Prior Plan, the Input Processor is responsible for translating it into the Partial Plan format.

2. Planning Controller

The Planning Controller contains the logic for control flow between the Mission Planners and Enhancement Planners. This could be a fixed sequence of function calls or a conditional, iterative logic based on Partial Plan features, Cost Function values, or other Planning Input data. Each Mission Planner and Enhancement Planner is a self-contained automated planning algorithm; the Planning Controller fulfills a higher-level management role. Since each subordinate planning algorithm may be very costly (theoretically, EXPTIME or even semidecidable depending on the planning approach), the Planning Controller should be as simple as possible.

The Mission Planners and Enhancement Planners are each allowed to use different objective functions and heuristics. The Planning Controller is responsible for dealing with these different measures. It may use a mathematical formula to combine all objective functions into a single, joint score, use only the User Cost Function, employ its own, unique objective function, or simply use the scores from the last-run Mission or Enhancement Planner. Whatever the method, the Planning Controller must select a single Partial Plan that achieves all Required Tasks and provide it to the Plan Compiler, if possible. This Partial Plan is called the *final plan*.

3. Annotated Mobility Graph

The *Annotated Mobility Graph* is a representation of the Map used for planning computations, in particular for Pathfinding. *Mobility* means that the representation must include information affecting the movement of units. We use the term *graph* rather than *grid* for generality, since both homogeneous (grid) and heterogeneous (mesh) discretizations of space are compliant with the framework. *Annotated* means that we put additional information in each node to use for tactical reasoning about units, such as visibility and risk.

The Annotated Mobility Graph is not included in the Planning Data because it is not expected to be useful across multiple scenarios. Its structure and contents may be dependent on the specific locations of units, which may change even during the execution

of a single scenario. However, caching of some elements of the Annotated Mobility Graph may be effective in eliminating re-work.

A node in the Annotated Mobility Graph is a location in discretized space—a point, area, or volume that may be occupied by one or more units. It is an implementation decision whether a single graph for all units templates or different graphs for different unit templates are necessary. The following types of data, or references to objects containing such data, may be included in the annotation of a node:

- The units that can target anything occupying this node
- The nodes or units that a unit in this node can target
- The terrain type of this node, used to look up rate of movement for a unit traversing it
- The longest lateral edge of the node (if nodes describe areas or volumes)
- The cost of traversing this node in terms of risk to enemy fires
- The set of no more than n -distant nodes forming a possible large-unit position area (for a positive integer n)

4. Pathfinding

Most conceivable tactical plans involve some kind of movement. Tactical movement planning is a relatively well-studied problem with some individually packaged solutions, so we put this part of the problem in its own component, called Pathfinding. Different planners may require different pathfinding approaches, but many of the algorithms and data structures are likely to be reusable between Plan Generator components. Pathfinding is expected to use the Annotated Mobility Graph as its search space, and it may use the derived movement models of the Planning Data for its calculations.

5. Partial Plans

Plan Generator components need a language to communicate with each other. The Task Dictionary is the lexicon of this language. Its grammar is the *partial plan representation* of the Plan Generator. The term comes from plan-space planning, but we

use it in a general sense here. In this framework, a partial plan is a set of task instances assigned to the units of the Friendly Force with a timing specification. It may contain gaps in time where no tasks are assigned to a particular unit, and it does not necessarily achieve all—or any—of the Required Tasks (yet). The parameters of each task instance may be partially bound. We assume that the Input Processor can insert the Required Tasks into Partial Plans and mark them as unfulfilled as a first step to processing.

A Plan Generator subcomponent that receives a partial plan as input must be able to operate on some of its tasks. It is not a requirement for every subcomponent to understand everything in the Task Dictionary, and subcomponents may be restricted to operate on a limited range of task parameter values. Each subcomponent should be able to ignore tasks that it cannot process in a partial plan.

The framework does not insist that planning follow the HTN paradigm; nevertheless, there is a natural hierarchical organization to any partial plan that results from the hierarchical organization of the units of the Friendly Force. Since the command hierarchy organizes all Friendly Force units into a tree, we can form an acyclic digraph of tasks in any partial plan by using the commanded-by relationships of each task's unit. In traditional HTN planning, a more descriptive tree of tasks (the decomposition tree) is formed by the HTN methods that expand or replace tasks with lower-level tasks. A Plan Generator component *may* use that approach, and *must* deal with the fact that tasks have an inherent hierarchical organization arising from their assigned units. Components may use additional layers of task hierarchy between the levels of the command hierarchy if desired—for example, a single task for a unit might be expanded into a sequence of tasks for that same unit, as one would expect in traditional HTN planning.

6. Mission Planner

A Mission Planner is a processing component that constructs or expands partial plans using a tactics configuration in order to achieve tasks. It reasons about units in relation to the Annotated Mobility Graph by using the Planning Data's derived models and the Enemy Force's Plan (if provided in the Planning Input). It may invoke the Pathfinding component to consider possible movement of units. Mission Planners are not

responsible for formatting output for the CSE. Their output is formatted as one or more partial plans that can be interpreted by other subcomponents of the Plan Generator.

The Plan Generator may contain more than one Mission Planner, and there is no requirement for any particular component to achieve any particular tasks as long as the overall system generates acceptable plans for the CSE. For example, we might use one type of Mission Planner for generating some high-level plans for a few different forms of maneuver, along with a separate Mission Planner for each form of maneuver. An alternative approach is to have a single Mission Planner that considers many different forms of maneuver. The creation of *multiple* Mission Planners may be called for in situations such as the following:

- When distinct classes of tasks require different domain logic
- Where independent parts of a plan can be processed in parallel, allowing a more efficient multithreaded approach to planning (in this case, some of the Mission Planners may be spawned instances of the same type)
- Where different types of tasks require different search algorithms for effective plan generation
- Where pre-existing Mission Planner components from other systems are brought together into a single system
- For a diverse CSE with many different unit types, such as a ground combat element and an air combat element
- For software engineering reasons, such as separating a large segment of code into separate manageable components
- For separation of work supported by different contractual agreements

By definition, a Mission Planner qualitatively achieves tasks. The formal meaning of “achieves” depends on the Partial Plan representation, but the point is that Mission Planners have a meaningful goal test for their planning branches. For example, if assaulting a geographical objective were one of the Required Tasks, then a Mission Planner component would generate a sequence of movement tasks that result in some Friendly Force units arriving at that objective. The units may only have a small probability of actually arriving there due to action by the Enemy Force, but a plan that does not move any units onto the objective can never fulfill the Required Task. An

individual Mission Planner need not achieve every task, but the Planning Controller is responsible for invoking multiple Mission Planners, if needed, to achieve all unfulfilled Required Tasks (or declare failure) and as many Preferred Tasks as possible. Some Mission Planners may not be invoked for every planning instance.

In battle planning, the challenge tends to be finding plans that work *well*, not just finding plans that work. The bulk of the Mission Planner's effort is likely to involve searching the vast space of "goal" plans for one with an acceptable score—or at least best discoverable score—according to some objective function. The objective function of each Mission Planner is expected to include the User Cost Function, but it is up to the planning developers to determine which terms are appropriate. Each Mission Planner is only expected to limit plan cost within the scope of its particular focus, which may be a warfighting function, a certain class of units, or some other aspect of planning. Furthermore, each Mission Planner may use a different objective function. It is not the Mission Planner's responsibility to combine its objective function or heuristics with those of other Mission or Enhancement Planners.

Military professionals should take care not to imply unintended meaning to the word *mission* in *Mission Planner*. There is no expectation that the Mission Planner generate a doctrinal mission statement. A single Mission Planner may generate and achieve additional "implied" tasks, including tasks for subordinate units if developers choose a multi-echelon approach. The framework is general enough to cover all sorts of military operations, not just attack maneuvers. In theory, a Mission Planner focused on tactical maneuver could generate an additional task to provide ammunition resupply at some point, and a separate Mission Planner focused on logistics could find ways to achieve the resupply task. In this case, the Planning Controller would need to deal with the possibility of the logistics planner failing (is some other maneuver plan supportable?). From the perspective of any Mission Planner, the "mission" is the qualitative achievement of its input tasks with the best possible objective function score.

7. Enhancement Planner

The only fundamental difference between a Mission Planner and an Enhancement Planner is that the latter does not qualitatively achieve tasks; it does not have a meaningful goal test. An Enhancement Planner expands or modifies a partial plan using a tactics configuration in order to improve a plan according to an objective function. An Enhancement Planner may be provided with any of the same data accessed by a Mission Planner, and it may invoke the Pathfinding component. In general, an Enhancement Planner assumes that its input partial plan already qualitatively achieves the tasks that matter to it. Enhancement Planners may expand existing tasks into multiple tasks, modify the parameters of existing tasks, replace existing tasks with new tasks, or delete tasks, but they are restricted from invalidating the qualitative achievement of a Required Task. Multiple Enhancement Planners may be required for the same reasons as listed above for multiple Mission Planners.

Readers with a military background may be tempted to think of Enhancement Planners as “support planners.” Indeed, Enhancement Planners can be used to model planning for some support functions (see Chapter IV), but that is not their only use. An Enhancement Planner could use an evolutionary algorithm to improve a maneuver plan, and a support function such as force protection may require the qualitative goal test of a Mission Planner. Benefits of differentiating between Mission Planners and Enhancement Planners include

- To help the Planning Controller prioritize computational resources. Mission Planners must achieve Required Tasks, but Enhancement Planners can be cut off at any time without invalidating partial plans.
- To support replanning. It may be possible to run Enhancement Planners to adjust to unexpected events without changing the qualitative structure of the overall plan.
- To employ a variety of cost functions that may not work well in a single search algorithm. For example, we lose our ability to perform uniform cost search if we add a nonincreasing term to the cost function, but we could use such a calculation in a separate Enhancement Planner.

Enhancement Planners may be thought of as plan “optimizers,” with the caveat that “optimal” plans are generally not achievable (at least not tractably). We avoid the

term “optimization planner” because “optimization” takes on a variety of meanings, and as a reminder that a numerically superior output is a worthless M&S product if the plan is not believable as a model of a human-generated plan.

We offer the Fire Support Planner (see Chapter IV) as an example of how this separation of effort can naturally support two key aspects of battle planning—maneuver versus fire support.

8. Plan Compiler

The Plan Compiler translates the Planning Controller’s final plan into a format that can be executed by the CSE. Translation is reliant upon Operational Actions, which by definition can be directly translated into actions understood by the CSE.

If the CSE cannot receive the entire plan at once—e.g., it understands only immediate actions—then it must be augmented with a plan execution component that stores future actions and invokes them at the planned time or event. We consider such a component outside the boundary of the ABPS.

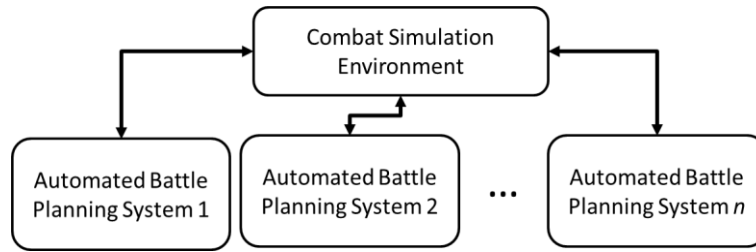
If the planning system is built for a specific CSE, it may be possible to use the Executable Plan format of the CSE as the partial plan representation. In this case, a Plan Compiler is not required. This special case also allows a Prior Plan to be provided with no need for translation by the Input Processor.

9. Separate Planning Systems

The framework allows the ABPS to generate a plan for any number of units residing at any number of different echelons using a single planning instance. We call this a *singleton planning system*. In real world (human) operations, planning requires n different planning processes for the n different units of a command. Planning is important for human commanders and staffs not just for the generation of the plan, but also for its contribution to the military design process (DOD 2011b), as discussed in Appendix C. Since real world planning proceeds in separate command silos (not to mention in warfighting function or staff section silos, despite constant effort to integrate), some incompatible results and re-work are inevitable in real world planning. Since our focus is

on tactical science, and we are not trying to represent human planning processes directly—only a semblance of their output—the framework allows all planning to occur within a single system that can continually maintain compatibility, balance resources, and compute more efficiently. Different tactical options at one unit can be evaluated by resolving plans for subordinate units for each option down to the lowest level of detail and computing their objective functions. In a sense, this is an alternative for the intuitive, experience-based estimates that human planners are constantly making about the activities of their subordinates, who may not yet even be aware of their forthcoming taskings.

However, some studies or training events may benefit from a separation of different units' planning efforts. A planning model that more closely resembles real world planning would have each unit generate a plan with tasks only for its immediate subordinates, then transmit those tasks explicitly through the CSE's communications component to other ABPSs—one for each subordinate unit. This design pattern is described by Pittman (2008) and implemented with two echelons in Sony's Killzone 2 and 3 games (Straatman et al. 2013). Our term for this approach is *separate planning systems* (Figure 20). For this approach to produce different results than a singleton planning system, different algorithms and derived models are needed: each ABPS instance must be able to evaluate its planning options without knowing for certain the detailed plans that its subordinates will choose. If we tried to use several copies of a singleton planning system, each unit at each echelon would produce a plan down to the smallest subordinate unit. Other than variations from communication delays or different Military Intelligence input, the result would be identical to one singleton planning system's plan (but at greater computational cost).



Although the units corresponding to each planning system may be hierarchically arranged (which is not shown here), they must communicate through the modeled environment.

Figure 20. Separate Planning Systems

An example of a study where separate planning systems would be needed is one that evaluates communication systems' impact on the planning process itself. If the information available to each unit for planning, including its Required Tasks from the commanding unit, must be transmitted through an imperfect communication system modeled in the CSE, then separate planning systems are a natural modeling approach.

E. SUMMARY

The planning framework described here supports a modular design for ABPSs, which in turn allows a division of labor in its development. The data-driven approach prescribed by the Planning Data may allow some components, such as the Planning Controller, Mission Planners, and Enhancement Planners, to be re-used across different CSEs (or at least simplify migration) because the CSE-specific details reside mostly in the Planning Data. A Partial Plan data structure should also be reusable as long as it encapsulates tasks, which are at least partially CSE-specific. Methods that work only with Operational Tasks stand a good chance of re-use, since Operational Tasks have no state-changing effect in the CSE. The Annotated Mobility Graph may also be reusable if references to units and terrain objects are carefully encapsulated. Pathfinding is already a well-understood field that sits naturally as a separate module.

Even when software cannot be reused without a migration or re-implementation effort, agreement on a modularization approach can help with the re-use and improvement upon ideas, algorithms, and data structures. It allows researchers and

developers to focus on specific areas, rather than starting from whole cloth with each new attempt at improving the status quo.

The framework makes few assumptions about the domain. It is general and modular enough to be applied to a wide range of combat M&S uses. As envisioned, a variety of different functions could be provided by a single ABPS. This could result in a complicated set of tasks and methods, which would naturally lead to organization by functional categories. An additional, generic category of tasks and methods would then be needed to bridge between these categories for full-spectrum operations.

We offer the framework as a starting point for the organization of research in this domain, but improvements on its structure, vocabulary, and assumptions are expected. As an initial test of its utility, we use it to guide the implementation found in Chapter V. The reader will find the language of this framework resident in the names of software elements there.

Although we assume the external world is a CSE—not the real world—an ABPS aligned with this framework could support real world operational planning in two ways. Both of the following approaches are only meant as supporting tools for human planners; we do not challenge the premise that commanders are ultimately responsible for their plan and its outcome.

- (1) If we replace the CSE with a real world command and control system (with a plan representation), then we have an operational planning system. Its simulation component would be limited to the capabilities of its derived models.
- (2) If we use an operational simulation environment for the CSE, then we have an operational planning system with a more robust and (probably) user-friendly simulation component. In this case, the simulation system serves as an additional plan evaluation component, and the quality of the output is partially dependent on the forecasting power of the simulation.

One of the benefits of agreement on a framework such as this one is a context for the insertion of new ideas. The specific new idea we offer here—described in the next chapter—is an approach to automated fire support planning. This is just one example of a

framework component. Other possible research lanes are discussed in the Future Work section of Chapter VII.

IV. THE FIRE SUPPORT PLANNER

A. OVERVIEW

This chapter presents the conceptual model for a fire support planner, which is meant to be used as an Enhancement Planner in an ABPS architecture (as explained in the previous chapter). The basic concepts discussed here were originally published by Harder and Darken (2016), but the treatment presented here is more thorough and uses some different notation.

We assume that an input plan, in some partial plan format, is provided to the fire support planner. Included in the input plan are the fire support units available to the fire support planner (see Availability Tasks, below). We also assume that the precise locations of Enemy Force units are available. The fire support planner updates the partial plan by assigning fire support tasks to available units with the objective of reducing the risk of casualties to the Friendly Force.

We make this general concept more specific in the rest of the chapter. We start by stating some assumptions and listing the type of information needed from the input plan. We then describe the data elements we use to measure risk and planning branches, followed by some operations needed to manipulate the data elements as we try to reduce the level of risk. With these tools in hand, we present a planning algorithm to conduct automated fire support planning. We follow this with an analysis of the time complexity of the algorithm. We conclude the chapter with some possible extensions to the method.

B. ASSUMPTIONS AND INPUT

The conceptual world of the fire support planner may differ from the external world (the combat model where the Executable Plan will be run), but we use certain kinds of abstractions to support automated planning. These are described here as assumptions.

The input plan could be a maneuver plan, i.e., movement and firing instructions for maneuver units conducting tactical operations. However, the fire support planner

treats the input plan quite generically; it could be any type of plan as long as it provides the information described below. Since the fire support planner produces aggressive movement and firing actions, the operational context of the situation should be appropriate for deliberate and intense application of firepower.

1. Time and Terrain

We assume a continuous model of time. Although we can translate a fixed time step model into a continuous time framework quite trivially, our method assumes that any time interval of nonzero duration can be subdivided at an arbitrary point between its endpoints, so a discrete implementation (even with floating point numbers) would need to deal with rounding errors.¹⁴

The *terrain* is a set of polygons in a Euclidean three-dimensional space. Each two-dimensional polygon of the terrain may inhabit a different plane, but may not be parallel with the y -axis. Terrain polygons may not overlap vertically: any line drawn parallel with the y -axis may intersect only one terrain polygon unless it passes through a single shared edge. The side of each terrain polygon facing towards the positive y direction (exactly or at an acute angle) is called its *surface*, and the *terrain surface* U is the set of all polygon surfaces of the terrain. The members of U are also called *nodes*. The terrain has no gaps, which means that all of its internal edges are shared, and the projection of the terrain onto the xz -plane is a convex polygon.¹⁵ The Annotated Mobility Graph subcomponent provides the terrain to the fire support planner. A visual example of a terrain is provided in Figure 21.

2. Units and Threat Annotations

A *unit* occupies an infinitesimal point on the terrain surface at any given time. Each unit is a member of the Friendly Force set F or the Enemy Force set E . Each $f \in F$ may move according to an equation of motion $\mathbf{x}(t)$, but \mathbf{x} is only valid over the subset of

¹⁴ The software implementation described in Chapter V is, of course, such a discrete implementation.

¹⁵ The terrain may be equivalent to the terrain representation of the combat model (the Map, in terms of the Conceptual Planning Framework), or it may be an approximation of it.

its domain where $\mathbf{x}(t)$ is on the terrain surface. Enemy Force units are assumed to be stationary (a reasonable assumption for a deliberate attack, since prepared fighting positions offer significantly better protection against incoming fire than hastily occupied positions). For each $f \in F, e \in E$ there exists a killing rate $\kappa_{ef} : S \rightarrow [0, \infty)$ where S is a conceptual world state (we will get more specific about which aspects of the world state matter). Units are assumed to have unlimited strength (i.e., they never run out of members), but the objective is to minimize the number of kills that the friendly force's units suffer.

The *threat annotation* is a function $E_U : U \rightarrow \wp(E)$, where $E_U(u)$ is the set of all Enemy Force units that can threaten a Friendly Force unit located anywhere on u , including its edges.¹⁶ The *threat-targeting annotation* is the corresponding function $f_U : U \rightarrow \wp(E)$, where $f \in F$, such that $f_U(u)$ is the set of all Enemy Force units that f can engage with effective fire support from u . The Annotated Mobility Graph provides the threat and threat-targeting annotations to the fire support planner.

3. Actions and Routes

We assume that the input plan is, or has enough information to produce, a set of *actions*. An action is a tuple (f, \mathbf{x}, t', t'') where

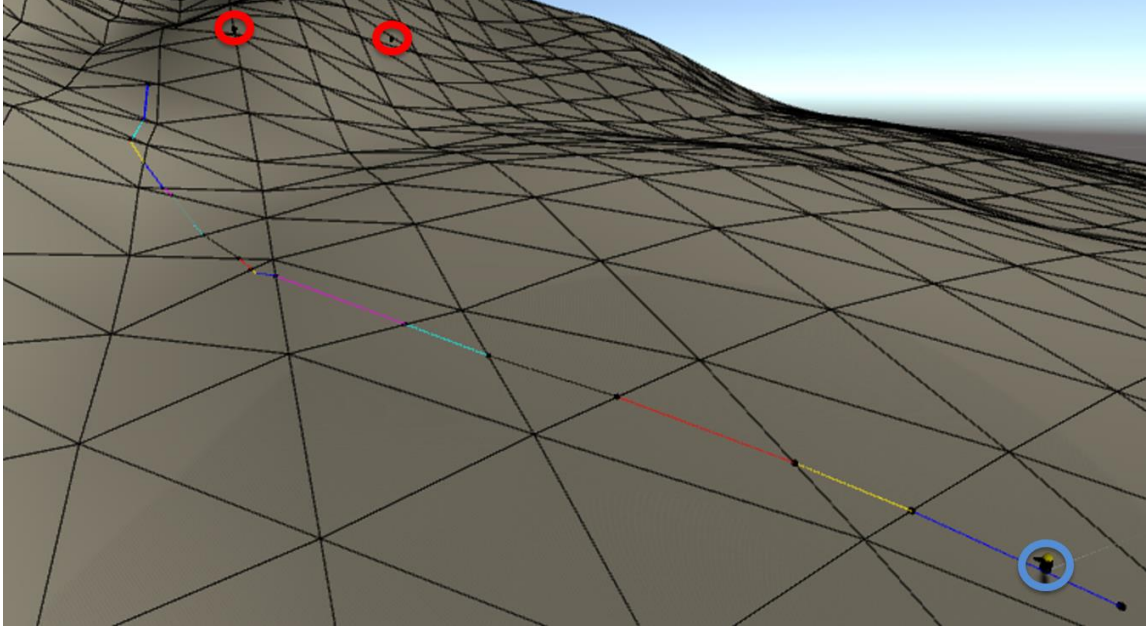
- $f \in F$
- $\mathbf{x} : \mathbb{R} \rightarrow \mathbb{R}^3$ is an equation of motion (\mathbf{x} may be a constant function if f is to remain stationary) on the terrain surface throughout $[t', t'']$; that is, $\forall t \in [t', t''] : \mathbf{x}(t)$ is a point on the terrain surface
- $t' \in [0, \infty), t'' \in (t', \infty)$ are the starting and ending times of the action (note that actions cannot have duration 0 since the definition enforces $t' \neq t''$)

Any tasks in the plan (i.e., operational tasks) that do not have this kind of location and movement information are ignored by the fire support planner. We do not require that

¹⁶ Whether the threat annotation is a precise shadow volume-based calculation (which would involve subdividing terrain polygons) or an approximation is left to the Annotated Mobility Graph. The fire support planner works as if the threat annotation is precise.

each $f \in F$ be completely specified with actions for the entire time interval of the input plan, but units with unspecified time intervals are considered invulnerable over those intervals—the planner will provision fire support only for the time intervals given in units' actions. No pair of actions in the input plan for the same unit may overlap in time, and if $(f, \mathbf{x}, t_0, t_1), (f, \mathbf{x}', t_1, t_2)$ are connected actions (the ending time of the first is the starting time of the second) for the same unit f , then we require that $\mathbf{x}(t_1) = \mathbf{x}'(t_1)$.

A *route* (Figure 21) is a sequence of connected actions $\pi = \langle (f, \mathbf{x}_1, t_1, t_2), (f, \mathbf{x}_2, t_2, t_3), \dots \rangle$ for a single unit. A *maximal route* is a route of maximum length; that is, the plan contains no actions for that unit that can be connected to the first or last element of the route. We use the alternate symbols $\mathbf{c} \rightarrow \mathbf{c}'$ for a route $\langle (f, \mathbf{x}_1, t_1, t_2), \dots, (f, \mathbf{x}_n, t_n, t_{n+1}) \rangle$, where $\mathbf{c} = \mathbf{x}_1(t_1), \mathbf{c}' = \mathbf{x}_n(t_{n+1})$, when we want to emphasize the starting and ending points \mathbf{c}, \mathbf{c}' . The *equation of motion for a route*, $\mathbf{x}_\pi(t)$, is the piecewise defined vector function for the location of that route's unit any time between the start time of its first action and the end time of its last action. Two routes π, π' with respective time intervals $[t_1, t_{n+1}], [t'_1, t_{m+1}]$ are equivalent if the actions of both are for the same unit, $t_1 = t'_1, t_{n+1} = t_{m+1}$ and $\forall t \in [t_1, t_{n+1}]: \mathbf{x}_\pi(t) = \mathbf{x}_{\pi'}(t)$. It is not necessary to have $n = m$.



The brown triangles represent terrain polygon surfaces (nodes). The icon circled in blue represents a Friendly Force unit, which by our definition only occupies a single point on the terrain surface (at its base). The multicolored line segments each represent the parts of the terrain surface that the unit will traverse as the result of actions, and the sequence of actions is a route. The icons circled in red are Enemy Force units.

Figure 21. A Unit Following a Route Along a Terrain Surface

Route subdivision is the process of creating an equivalent route by replacing an action with two actions describing the same movement as the first. We use the bar symbol “|” to specify subdivision at a particular time point. Given route $\pi = \langle w_1, \dots, (f, \mathbf{x}_{i-1}, t_{i-1}, t_i), (f, \mathbf{x}_i, t_i, t_{i+1}), (f, \mathbf{x}_{i+1}, t_{i+1}, t_{i+2}), \dots, w_n \rangle$ and $t \in \mathbb{R}$, where $t_i < t < t_{i+1}$, define

$$\pi | t = \langle w_1, \dots, (f, \mathbf{x}_{i-1}, t_{i-1}, t_i), (f, \mathbf{x}_i, t_i, t), (f, \mathbf{x}_i, t, t_{i+1}), (f, \mathbf{x}_{i+1}, t_{i+1}, t_{i+2}), \dots, w_k \rangle \quad (1)$$

If $t_i < t < t_{i+1}$ does not hold for any $w_i \in \pi$ (either t is outside the route’s whole time interval or t is a start or end time of one of its actions), then $\pi | t = \pi$.

4. Availability Tasks

An *availability task* is a tuple $a = (f, \mathbf{c}, t', t'')$ where

- $f \in F$
- \mathbf{c} is a point on the terrain surface
- $t' \in [0, \infty), t'' \in [t', \infty)$ are start and end times as in an action, except that 0-duration availability tasks are possible (for mathematical convenience)

The meaning of a is that f is provisionally planned to be at \mathbf{c} at time t and available for fire support tasking during interval $[t', t'']$. Availability tasks are not actions, but they are provided with the input as an indication of the resources available to the fire support planner. We think of each distinct a as a separate fire support planning resource.

5. Suppression

We described the military concept of suppression in Chapter II. We define suppression here, a bit more precisely, as the temporary reduction of the killing rate of a unit. In the terminology of Lanchester equations, this means a penalty to the efficiency coefficient. A single suppression action is assumed to have a proportional effect on all targets threatened by the suppressed unit during the suppression time interval. We assume that suppression causes no casualties to the threat.

Given a killing rate $\psi \in [0, \infty)$, which is measured in expected number of casualties per unit time, we represent suppression as a function $\phi: [0, \infty) \rightarrow [0, \infty)$ where $\phi(\psi) < \psi$. Since a killing rate may vary over time, in particular as the distance between threats and their targets changes, ψ is not constant but a function of time. Thus, ϕ is a transformation of ψ such that $\phi(\psi(t)) < \psi(t)$. We require any pair of killing rate transformations ϕ and ϕ' to be commutative; that is, $\phi(\phi'(\psi(t))) = \phi'(\phi(\psi(t)))$. The need for commutativity will become more apparent below, as we describe the plan-space planning approach of the algorithm.

C. DATA ELEMENTS

In addition to the availability tasks just described, a planning node contains representations of tactical risk and new tasks for the fire support resources. We define these, and then formalize the notion of a planning node.

1. Risk Object Hierarchy

We use the routes of units to create representations of the risk to those units. This representation is a four-tiered model (Figure 22), which we describe from the bottom up in the following subsections. In the most basic sense, a *risk set* is a collection of *risk intervals*, each of which has a sequence of *risk subregions*, each of which has a sequence of *risk segments*.

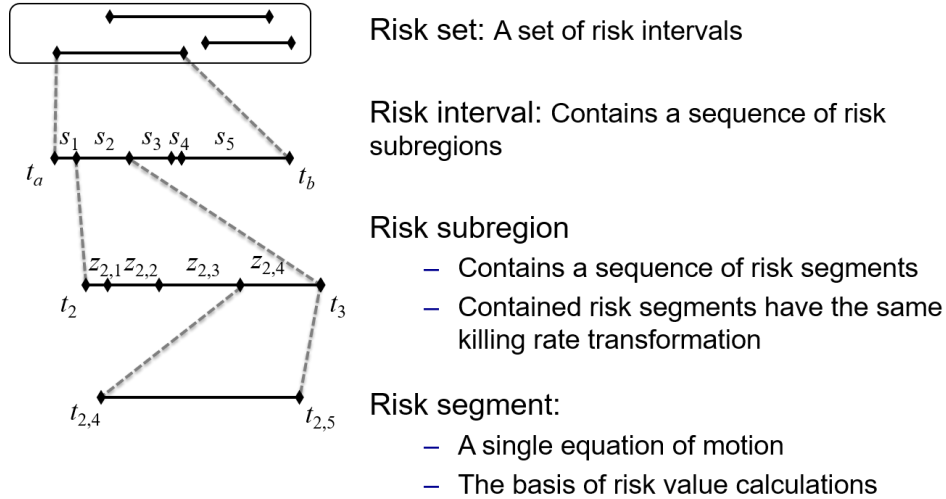


Figure 22. The Risk Object Hierarchy

a. Risk Segments

A *risk segment* is a tuple $(\psi, \mathbf{x}, t', t'')$ where

- $\psi(t)$ is a killing rate valid over $[t', t'']$
- \mathbf{x} is an equation of motion along the terrain surface during $[t', t'']$ (exactly like an action)

- t', t'' are start and end times (exactly like an action)

A risk segment represents the risk from a single Enemy Force unit to a single Friendly Force unit during an action. Given an action $(f, \mathbf{x}_1, t_1, t_2)$ constrained to a single node (terrain polygon surface) u , we can construct a risk segment for each $e \in E_U(u)$ using the killing rate κ_{ef} . The resulting risk segment is $(K \circ \kappa_{ef}, \mathbf{x}_1, t_1, t_2)$, where K is a transformation given by derived detection models and combat effects models, if needed (see Chapter III, Section B.2). Assuming that we can look up $K \circ \kappa_{ef}$ in constant time given e and f (for example, in a hash table) and both $K \circ \kappa_{ef}$ and \mathbf{x}_1 are limited to a constant number of symbols, we can construct a risk segment (for a single Enemy Force unit) from an action in constant time.

If an action w' on a route $\pi = \langle w_1, \dots, w', \dots, w_n \rangle$ traverses $k > 1$ nodes, we can use subdivisions to produce the equivalent route $\pi' = \langle w_1, \dots, w'_1, w'_2, \dots, w'_k, \dots, w_n \rangle$, where each w'_i is constrained to a single node, to construct risk segments. Similarly, if $w' \in \pi$ crosses boundary points of $K \circ \kappa_{ef}$ —for example, if κ_{ef} is a piecewise defined function¹⁷—we can compute the arrival time t at each boundary point and subdivide π at each t .

The *risk value function* Ψ gives the expected losses for a risk segment. Given $z_{i,j} = (\psi_{i,j}, \mathbf{x}, t', t'')$, we define

$$\Psi(z_{i,j}) = \int_{t'}^{t''} \psi_{i,j}(t) dt \quad (2)$$

(Indices i and j are explained in the following sections.) Let T_Ψ be the upper-bound computational cost to calculate $\Psi(z_{i,j})$. This cost depends on implementation details, but we provide one analytical solution in Section H, below, that runs in constant time (assuming a fixed bit length on its numerical inputs). If we do not have the analytical

¹⁷ Our implementation deals with a piecewise-defined probability of hit. See Chapter V.

solution for $\Psi(z_{i,j})$, we can use numerical integration, but this may incur a significantly greater cost depending on the level of dithering.

One interpretation of the risk value calculation is that we are using spatial Lanchester (1916) equations to generate expected values for planning. We have only one side of the Lanchester system of differential equations because we are not applying a killing rate to the Enemy Force units. Also, note that we are accruing losses rather than reducing strength, as the traditional formulation of Lanchester equations do.

b. Risk Subregions

A *risk subregion* is a tuple (ϕ_i, t', t'', z_i) where ϕ_i, t', t'' are, respectively, a killing rate transformation, a start time, and an end time. z_i is a connected sequence of risk segments $\langle (\phi_i \circ \psi_{i,1}, \mathbf{x}_{i,1}, t', t_{i,2}), (\phi_i \circ \psi_{i,2}, \mathbf{x}_{i,1}, t_{i,2}, t_{i,3}), \dots, (\phi_i \circ \psi_{i,k}, \mathbf{x}_{i,k}, t_{i,k}, t'') \rangle$ in the same sense that a route is a connected sequence of actions. Note that every killing rate in the sequence is the composition of ϕ_i (the killing rate transformation of the subregion) with a killing rate $\psi_{i,j}$.

The *equation of motion for a risk subregion* is analogous to the corresponding term for routes. Given a risk subregion s_i , equation of motion $\mathbf{x}_{s_i}(t)$ is the piecewise defined vector function giving a location at any time between its start and end times, as specified by its contained risk segments. Two risk subregions $s_1 = (\phi_1, t'_1, t''_1, z_1), s_2 = (\phi_2, t'_2, t''_2, z_2)$ are equivalent if $\phi_1 = \phi_2, t'_1 = t'_2, t''_1 = t''_2$, and $\forall t \in [t'_1, t''_1]: \mathbf{x}_{s_1}(t) = \mathbf{x}_{s_2}(t)$. This is the same notion of equivalence we use for routes (except that risk subregions have killing rates instead of units).

To construct a risk subregion s_i from a connected sequence of risk segments $z_i = \langle (\psi_{i,1}, \mathbf{x}_{i,1}, t_{i,1}, t_{i,2}), \dots, (\psi_{i,k}, \mathbf{x}_{i,k}, t_{i,k}, t_{i,k+1}) \rangle$, we write $s_i = (I, t_{i,1}, t_{i,k+1}, z_i)$, where I is the identity function. More interesting killing rate transformations are introduced in later

sections. A risk subregion only has a constant number of symbols in addition to those of its contained risk segments, so the time complexity of risk subregion creation is $O(|z_i|)$.

The risk value of a risk subregion $s_i = (\phi_i, t', t'', \langle z_{i,1}, z_{i,2}, \dots, z_{i,k} \rangle)$ is defined as the sum of the risk values of its risk segments (note that we overload the symbol Ψ):

$$\Psi(s_i) = \sum_{j=1}^k \Psi(z_{i,j}) \quad (3)$$

It follows immediately that a risk subregion's risk value can be calculated in $O(|z_i|T_\Psi)$ time.

Any pair of equivalent risk subregions has the same risk value because the definite integral of a function (appearing in $\Psi(z_{i,j})$) is the same for any valid piecewise computation.

(Index i is explained in the next section.)

c. *Risk Intervals*

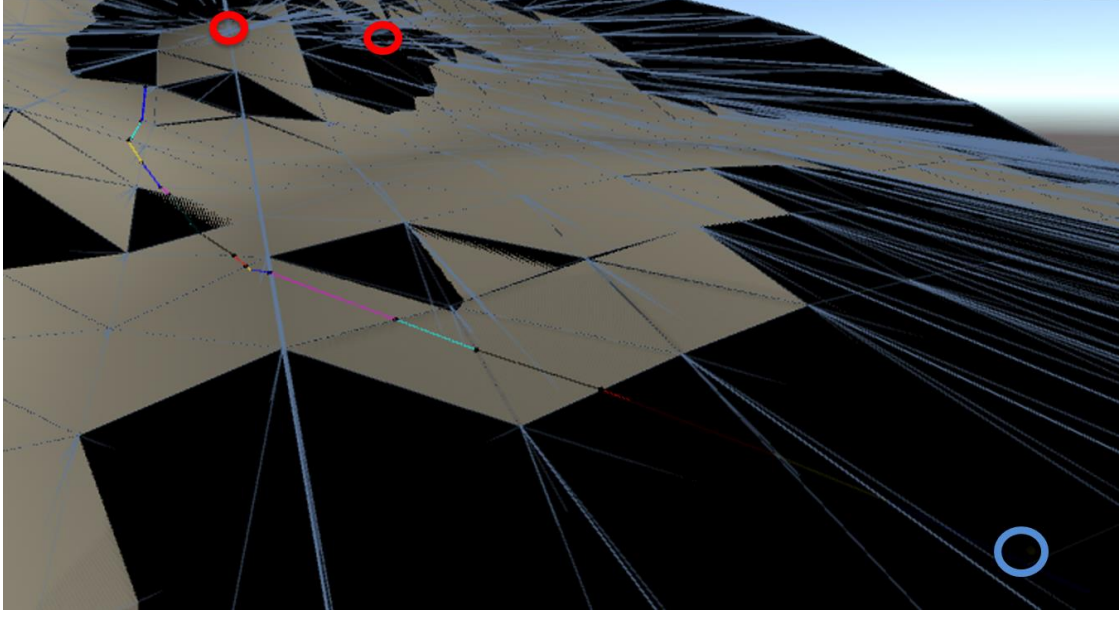
A *risk integral* is a tuple (e, f, t', t'', s, Φ) , where

- $e \in E$
- $f \in F$
- t', t'' are start and end times (as in the preceding definitions)
- s is a connected sequence of risk subregions $\langle s_1, \dots, s_k \rangle = \langle (\phi_1, t_1, t_2, z_1), \dots, (\phi_k, t_k, t_{k+1}, z_k) \rangle$
- $\Phi = \sum_{i=1}^k \Psi(s_i)$ is the sum of the risk values of the contained risk subregions

Each risk interval corresponds to a route with continual threat (nonzero killing rate) from a single Enemy Force unit. To construct a risk interval, choose $e \in E$ and a route $\pi = \langle w_1, \dots, w_n \rangle = \langle (f, \mathbf{x}_1, t_1, t_2), \dots, (f, \mathbf{x}_n, t_n, t_{n+1}) \rangle$ where $\forall w_i \in \pi : e \in E_U(w_i)$. First,

construct a risk segment from each w_i . Since the actions of π are a connected sequence, these new risk segments are a connected sequence; call it z_1 and create a risk subregion s_1 from it (as described in the previous section). Finally, create the risk interval $(e, f, t_1, t_{n+1}, \langle s_1 \rangle, \Psi(s_1))$. The time complexity of this construction is $O(nT_\Psi)$, proportional to the number of actions in π . Note that new risk intervals constructed in this way have a risk subregion sequence with just one element (s_1 , whose risk segment sequence contains n elements). In later sections, we describe how and why the risk subregion sequence is modified to include more elements.

If a route π traverses some nodes u where $e \notin E_U(u)$, then—by definition—we cannot create a single risk interval for f and e that covers the whole route. For convenience, let $U_e = \{u \in U \mid e \in E_U(u)\}$. To create risk intervals for π , we must produce an equivalent route π' in which no single action traverses nodes in both U_e and $U - U_e$. We can then use routes that are subsequences of π' , traversing only elements of U_e , to create multiple risk intervals for the portions of π threatened by e . This situation is depicted in Figure 23.



This is the same situation from the previous figure, but we have blacked out all of the polygons that do not have the Enemy Force unit on the left in their threat annotations. We would need at least two risk intervals to cover all parts of the route threatened by this unit (the multicolored line segments still visible).

Figure 23. Threatened Portions of a Route

Although it is possible to contrive an action that crosses back and forth between members of U_e and $U - U_e$ many times, it is reasonable to assume that such complicated equations of motion are not used in any CSE of interest. Let C be a constant such that no single action traverses more than C terrain nodes. Then an upper bound for the number of actions required for π' —the same route, equivalent to π , mentioned above—is Cn . No pair of maximal subsequences of π' may share an action (because they are maximal), so the number of steps to construct all risk intervals (for e) for these maximal routes is $O(CnT_\Psi) = O(nT_\Psi)$.

Naturally, the risk value of a risk interval is just the value stored in its final element. For $r = (e, f, t', t'', \langle s_1, \dots, s_k \rangle, \Phi)$,

$$\Psi(r) = \Phi = \sum_{i=1}^k \Psi(s_i) \quad (4)$$

When constructing r , Φ can be calculated in $O(nT_\Psi)$ time, where n is the number of risk segments (not subregions) contained in r .

d. Risk Sets

A *risk set* is simply a set of risk intervals. At the start of fire support planning, we create a risk set for the whole input plan. We then modify this set, by adding or replacing risk intervals (removing an old version and inserting an updated version). We often need to deal with a subset of a risk set R such as the following examples:

- All risk intervals overlapping time interval $[10, 53.2]$:
 $\{(e, f, t_a, t_b, s, \Phi) \in R \mid t_a \leq 53.2 \wedge t_b \geq 10\}$
- All risk intervals resulting from unit e : $\{(e', f, t_a, t_b, s, \Phi) \in R \mid e' = e\}$

Consider the construction of maximal risk intervals for a set of routes W , where each $w \in W$ has at most k actions and each action is constrained to one terrain node (as described in the Risk Segments section). In the worst case, every $e \in E$ threatens every action of w , so we need k risk segments $z_{i,j}$ for each pair $(e, w) \in E \times W$. Each such $z_{i,j}$ appears in only one risk interval for e . In the pseudocode found below, we represent this type of risk set construction with the subroutine-like notation $\text{BuildRiskIntervals}(W)$. Based on the preceding arguments, its running time is

$$T_B(|E|, |W|, k) = O(|E||W|kT_\Psi) \quad (5)$$

An upper bound for the size of the constructed risk set R is

$$N_B(|E|, |W|, k) = O(|E||W|k) \quad (6)$$

The risk value of a risk set R is the sum of the risk values of its members:

$$\Psi(R) = \sum_{r \in R} \Psi(r) \quad (7)$$

Since risk interval tuples already contain their computed risk values, we can calculate the risk value of a risk set in time

$$T_{\Psi(R)} = O(|R|) \quad (8)$$

Risk sets are the top tier of the risk object hierarchy depicted in Figure 22.

2. Fire Support Tasks

A fire support task is a tuple $(f, \pi, \mathbf{c}, \phi, e, t_1, t_2, t_3)$ where

- $f \in F$
- π is a route $\langle (f, \mathbf{x}_1, t_1, t_2), \dots, (f, \mathbf{x}_n, t_n, t_{n+1}) \rangle$ or the empty sequence $\langle \rangle$ (also called a *null route*)
- $\mathbf{c} = \mathbf{x}_n(t_{n+1})$ is a point on the terrain surface
- ϕ is a killing rate transformation (the same concept presented for risk subregions)
- $e \in E$
- $t_1 \in [0, \infty), t_2 \in [t_1, \infty), t_3 \in (t_2, \infty)$ are, respectively, the start time, effects time, and end time

A fire support task is an order for f to move along route ρ to firing point \mathbf{c} during $[t_1, t_2]$, and then fire on e during $[t_2, t_3]$. π includes any preparatory actions, such as emplacing and preparing weapon systems and even firing the first few volleys. In this conceptual model, the effects of f 's fire are limited to the reduction of e 's killing rate by ϕ , which is a derived combat effects model. $[t_2, t_3]$ is the interval of the effects, meaning that ϕ transforms the killing rate of e only during this time.

For the sake of simplicity, we assume that the end time of the suppression effects (t_3) is equal to the end time of the fire support task (i.e., the time at which f may start executing another task). This is a conservative assumption, since fired projectiles may still be en route to the target at that moment and e may remain suppressed for a short time after the last impacts.

3. Planning Nodes

A planning node is a tuple (A, P, R, W) where

- A , the *availability set*, is a set of availability tasks (unspent fire support resources)
- P , the *potential task set*, is a set of fire support tasks
- R is a risk set
- W is a partial plan

W contains all of the actions of the input plan as well as any fire support tasks added by the fire support planner.

Let W_f be the actions and fire support tasks in W assigned to unit f . We require each $a = (f, \mathbf{c}, t', t'') \in A$ to be disjoint in time with all members of W_f since availability tasks advertise the untasked time intervals of each unit.

The members of P are the planning branches; none of these fire support tasks have (yet) been added to W . Members of P do not need to be disjoint in time with each other, but for each $w = (f, \mathbf{c} \rightarrow \mathbf{c}', \phi, e, t_1, t_2, t_3) \in P$, there must be exactly one $(f, \mathbf{c}, t', t'') \in A$ bounding its time interval $([t_1, t_3] \subseteq [t', t''])$, and the location of this availability task must be the start location of w 's route (\mathbf{c}).

R is initialized by constructing all maximal risk intervals from the maximal routes of the input plan. We modify R with the operations described below each time we choose a planning branch. Once R has been updated for the choice of a new fire support task, the cost of the current partial plan is $\Psi(R)$.

D. OPERATIONS

Operations produce modified data elements from existing ones or retrieve previously constructed ones.

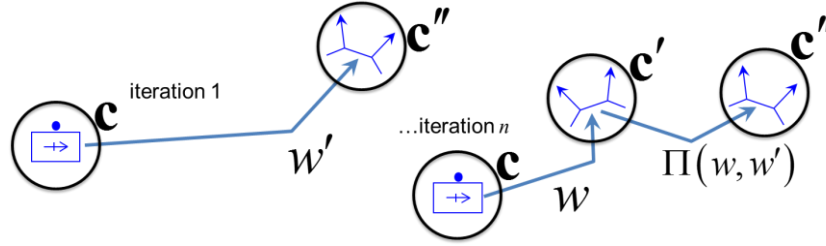
1. Immediately Following Task

Given partial plan W and availability task $a = (f, \mathbf{c}, t', t'')$, the *immediately following fire support task* $\varpi(W, a)$ is $(f, \pi, \mathbf{c}', \phi, e, t'', t_2, t_3) \in W$; that is, the fire support task in W for the same unit starting at the end time of a . If no such fire support task exists, to include the case where an action $(f, \mathbf{x}, t'', t_2)$ rather than a fire support task immediately follows a , then $\varpi(W, a)$ is undefined.

The computational cost of ϖ is $O(1)$ if the mappings from availability tasks to immediately following tasks are stored in a hash table for any partial plan W and updated any time one of these elements is replaced in A or W .

2. Route Replacement

The fire support planning algorithm works in plan-space, which means that each new task may be added over any available time interval in the plan. In particular, it is possible to add a fire support task for some unit f prior to another fire support task for f . This results in the situation depicted in Figure 24



In the first iteration, w' is added to the plan. In iteration n , w needs to be added to the plan, invalidating the route $\mathbf{c} \rightarrow \mathbf{c}''$ of w' . A new task $\Pi(w, w')$ with route $\mathbf{c}' \rightarrow \mathbf{c}''$ is needed.

Figure 24. Route Replacement

Assume that a new fire support task $w = (f, \mathbf{c} \rightarrow \mathbf{c}', \mathbf{c}', \phi, e, t_1, t_2, t_3) \in P$ needs to be added to partial plan W , and let $a = (f, \mathbf{c}, t', t'') \in A$ be an availability task bounding its

time interval $([t_1, t_3] \subseteq [t', t''])$. Also, assume $w' = \varpi(W, a) = (f, \mathbf{c} \rightarrow \mathbf{c}'', \mathbf{c}'', \phi', e_2, t_4, t_5, t_6)$ is the fire support task immediately following w . Define $\Pi(w, w') = (f, \mathbf{c}' \rightarrow \mathbf{c}'', \mathbf{c}'', \phi', e_2, t'_4, t_5, t_6)$ as the *route replacement function*. We can use the new task $w'' = \Pi(w, w')$ as a replacement for w' in W . w'' has the following differences from w' :

- The route $\mathbf{c} \rightarrow \mathbf{c}''$ is replaced by $\mathbf{c}' \rightarrow \mathbf{c}''$ because the former becomes invalid (it begins at the wrong location) once w is planned before w'
- The start time t_4 is replaced by t'_4 because $\mathbf{c}' \rightarrow \mathbf{c}''$ probably needs to begin at a different time in order for f to reach \mathbf{c}'' by t_5

If $t'_4 < t_3$, then $W - \{w'\} \cup \{w, w''\}$ is not a valid plan, but this concern is external to Π .

The computational cost of Π is dominated by the pathfinding cost to create $\mathbf{c}' \rightarrow \mathbf{c}''$, which we discuss below. We assume the pathfinding step provides the new start time t'_4 . The other steps to create w'' are copying operations for its elements, requiring just $O(1)$ time.

3. Availability Task Remainders

Assume that a new fire support task $w = (f, \mathbf{c} \rightarrow \mathbf{c}', \mathbf{c}', \phi, e_1, t_1, t_2, t_3) \in P$ needs to be added to partial plan W , and let $a = (f, \mathbf{c}, t_0, t_4) \in A$ be an availability task bounding its time interval $([t_1, t_3] \subseteq [t_0, t_4])$. Define the *availability task remainders function* as

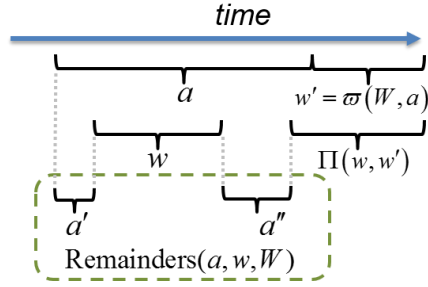
$$\text{Remainders}(a, w, W) = \{(f, \mathbf{c}, t_0, t_1), (f, \mathbf{c}', t_3, t')\}$$

$$\text{where } t' = \begin{cases} t_4, & \text{if } \varpi(W, a) \text{ is undefined} \\ t'_4, & \text{if } \Pi(w, \varpi(W, a)) = (f, \pi', \mathbf{c}'', \phi', e', t'_4, t_5, t_6) \wedge t_3 \leq t'_4 \end{cases}$$

If $t_3 > t'_4$, then $\text{Remainders}(a, w, W)$ is undefined.

The remainders function lets us account for the time interval of w , which can no longer be advertised as “available,” in A . This operation is simply an extension of the

following set operation over $\mathbb{R} : [t_0, t_4] - [t_1, t'] = [t_0, t_1] \cup [t_3, t']$, with the additional machinery to handle route replacement (Π). In the fire support planning algorithm, each choice of a new task w requires an update $A \leftarrow A - \{a\} \cup \text{Remainders}(a, w, W)$. Any resulting availability tasks of zero duration ($t_0 = t_1$ or $t_3 = t'$) serve no further purpose and may be discarded, but we do not handle that detail in this exposition.



This example shows the case where $\varpi(W, a)$ is defined and $\text{Remainders}(a, w, W) = \{a', a''\}$.

Figure 25. Availability Task Remainders

It is helpful, especially when implementing the immediately following task, to realize that an availability task can never be immediately following another availability task (for the same unit). If such a situation exists in the original availability set provided as input, we can replace any connected pair of availability tasks with a single one over the joint time interval. Given an initial setup with no connected availability tasks for any unit, let us prove that the property holds under repeated updates $A \leftarrow A - \{a\} \cup \text{Remainders}(a, w, W)$. We use strings over the alphabet $\{a, w\}$ to represent sequences of tasks for a single unit, where w represents any fire support task and a represents any availability task. Before planning occurs, the string a represents the sequence containing just one availability task. Updates are represented by the simple context-free grammar $\{a \rightarrow awa\}$. In the case of route replacement, let us temporarily add symbols w', w'' to the alphabet. Given a string $Aaw'aB$ where A and B are substrings

and $\Pi(w, w') = w''$, the updated task sequence is $Aawaw''aB$. This is no different with respect to adjacency of availability and fire support tasks than $AawawaB$, so we proceed using only w to represent fire support tasks.

We show by strong induction that there is no way to produce the substring aa using our simple grammar. For the base case, the initial string a trivially meets the condition. Assume the condition holds for all generated strings of length 1 to $k-1$. A string of length k must be of the form AwB , where A and B are each strings generated from a single a or the empty string. Both A and B are of length $k-1$ or less, so both meet the condition by induction, or trivially as the empty string. The w of AwB separates any pair of as drawn one-each from A and B , so AwB also meets the condition. Even if we remove any number of as (such as those corresponding to availability tasks with duration 0), the shortened string may have adjacent ws but no adjacent as . ■

Assuming the replaced route has already been created, the Remainders function needs only create two new availability tasks in $O(1)$ time.

4. Risk Subdivision

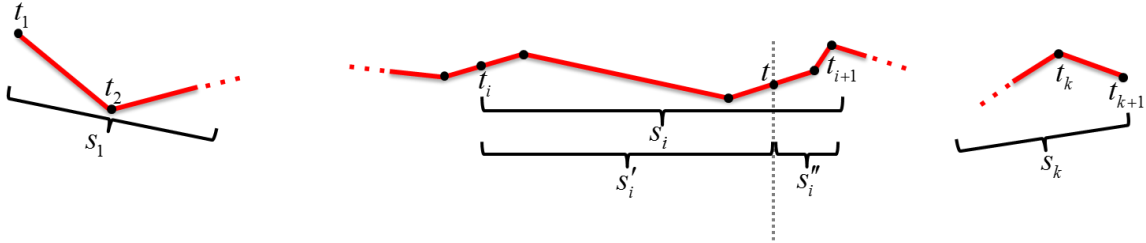
Given a connected sequence of risk subregions $s = \langle s_1, \dots, s_{i-1}, s_i, s_{i+1}, \dots, s_k \rangle$ and a time point t , *risk subdivision* is defined as

$$s | t = \begin{cases} \langle s_1, \dots, s_{i-1}, s'_i, s''_i, s_{i+1}, \dots, s_k \rangle, & \text{if } \exists s_i = (\phi_i, t_i, t_{i+1}, z_i) \in s : t_i < t < t_{i+1} \\ s, & \text{otherwise} \end{cases} \quad (9)$$

where

$$\begin{aligned}
s'_i &= (\phi_i, t_i, t, z'_i), \quad s''_i = (\phi_i, t, t_{i+1}, z''_i), \\
z_i &= \langle z_{i,1}, \dots, (\psi_i, \mathbf{x}_{i,j-1}, t_{i,j-1}, t_{i,j}), (\psi_i, \mathbf{x}_{i,j}, t_{i,j}, t_{i,j+1}), \dots, z_{i,n} \rangle, \\
t_{i,j} &\leq t < t_{i,j+1}, \\
z'_i &= \begin{cases} \langle z_{i,1}, \dots, (\psi_i, \mathbf{x}_{i,j-1}, t_{i,j-1}, t_{i,j}) \rangle, & \text{if } t_{i,j} = t \\ \langle z_{i,1}, \dots, (\psi_i, \mathbf{x}_{i,j}, t_{i,j}, t) \rangle, & \text{otherwise} \end{cases} \\
z''_i &= \begin{cases} \langle (\psi_i, \mathbf{x}_{i,j}, t_{i,j}, t_{i,j+1}), \dots, z_{i,n} \rangle, & \text{if } t_{i,j} = t \\ \langle (\psi_i, \mathbf{x}_{i,j}, t, t_{i,j+1}), \dots, z_{i,n} \rangle, & \text{otherwise} \end{cases}
\end{aligned}$$

Risk subdivision is similar to route subdivision, but we must additionally break apart the connected sequence of risk segments z_i . This requires at most one risk segment $(\psi_i, \mathbf{x}_{i,j}, t_{i,j}, t)$ to be split into $(\psi_i, \mathbf{x}_{i,j}, t_{i,j}, t)$ and $(\psi_i, \mathbf{x}_{i,j}, t, t_{i,j+1})$, which appear at the end and beginning of z'_i and z''_i , respectively. Risk subdivision produces an equivalent sequence of risk subregions $(s | t \equiv s)$, which therefore has the same risk value. Figure 26 provides a visual example.



Each red line segment between a pair of black dots represents a risk segment. The dotted vertical line represents the time point t at which the subdivision occurs. Risk subregion s_i is replaced by s'_i and s''_i .

Figure 26. Risk Subdivision

The simplest possible algorithmic approach to risk subdivision is a linear search over the risk subregion sequence to find s_i (if it exists), followed by a linear search over risk segment sequence z_i to find $(\psi_i, \mathbf{x}_{i,j}, t_{i,j}, t)$. This implies an upper bound of

$O(|s| + |z_i|)$. Given a linked list implementation of sequences, we need only constant time to perform subdivision as an update to an existing risk subregion sequence (after we have found the correct risk subregion and risk segment). To generate an entirely new copy of s requires time $O(m)$, where m is the total number of risk segments in members of s .

It is convenient to have a subroutine that performs two risk subdivision updates (rather than copies) on the risk subregion sequence of a risk interval. Given $r = (e, f, t', t'', s, \Phi)$ and $t_1, t_2 \in [0, \infty)$, define $\text{Subdivide}(r, t_1, t_2)$ to update s such that s is equivalent to $s | t_1 | t_2$.

5. Risk Reduction

Given a fire support task $w = (f, \pi, \mathbf{c}, \phi_w, e, t_1, t_2, t_3)$ and a risk interval $r = (e', f', t_a, t_b, s, \Phi)$, the *risk reduction function* is defined as

$$\lambda(r, w) = \begin{cases} (e', f', t_a, t_b, s', \Phi'), & \text{if } e = e' \wedge [t_2, t_3] \cap [t_a, t_b] \neq \emptyset \\ r, & \text{otherwise} \end{cases} \quad (10)$$

where

$$\begin{aligned} [t_{\min}, t_{\max}] &= [t_2, t_3] \cap [t_a, t_b], \\ s | t_{\min} | t_{\max} &= \langle (\phi_1, t_1, t_2, z_1), \dots, (\phi_i, t_{\min}, t_{i+1}, z_i), \dots, (\phi_j, t_j, t_{\max}, z_j), \dots, (\phi_n, t_n, t_{n+1}, z_n) \rangle, \\ s' &= \langle (\phi_1, t_1, t_2, z_1), \dots, (\phi_w \circ \phi_i, t_{\min}, t_{i+1}, z_i), \dots, (\phi_w \circ \phi_j, t_j, t_{\max}, z_j), \dots, (\phi_n, t_n, t_{n+1}, z_n) \rangle, \\ \Phi' &= \sum_{s_m \in s'} \Psi(s_m) \end{aligned}$$

$[t_{\min}, t_{\max}]$ is the time interval shared by r and the suppression effects of w . λ subdivides s at t_{\min}, t_{\max} and applies w 's killing rate transformation ϕ to each risk subregion bounded by $[t_{\min}, t_{\max}]$. The killing rate of each contained risk segment of these affected subregions changes from $\phi_m \circ \psi$ to $\phi_w \circ \phi_m \circ \psi$. This is how we record the suppression effects of w (ϕ_w reduces the killing rate of the target while f is firing on it). The new risk value Φ' is, as for any risk interval, the sum of the risk values of its contained risk

subregions (after ϕ_w has been applied). We only need to calculate the risk values of the subsequence s'_Δ of s' bounded by $[t_{\min}, t_{\max}]$ since $\Phi' = \Phi - \Psi(s_\Delta) + \Psi(s'_\Delta)$ (where s_Δ is the same subsequence prior to the addition of ϕ), requiring time T_Ψ for each contained risk segment calculation.

The time complexity of a risk reduction is the same for an update or copy operation. For an update, we perform the subdivisions in time $O(|s| + |z_i|)$, then add ϕ to the affected risk subregions and their contained risk segments in time and calculate Φ' in $O(kT_\Psi)$, where k is the total number of risk segments in the members of s' (in the worst case, every member of s' is affected). To perform the copy, we need $O(k)$ steps to create all of the copied risk segments after performing the risk value computations. Either way, the time complexity is

$$T_\lambda(k) = O(kT_\Psi) \quad (11)$$

Since λ requires only 2 risk subdivisions, the maximum number of risk segments in $\lambda(r, w)$ is

$$N_\lambda(r) = k_z + 2 \quad (12)$$

where k_z is the total number of risk segments in the risk subregions of r .

6. Task Scores and Affected Risk Sets

Since the objective function for a planning node (A, P, R, W) is $\Psi(R)$, the score of a potential task $w \in P$ is the difference $\Psi(R) - \Psi(R')$, where R' is the risk set that would result if w were added to W (to include route replacement, if needed—we present those details in the pseudocode for the complete algorithm, below). Note that we represent “changed” risk intervals (due to risk reduction) by removal and replacement rather than updates. We use a hash table, called Δ , to store the current potential tasks scores—i.e. $\Delta(w) \in \mathbb{R}$ is the score of potential task w . If $\Delta(w) > 0$, then w improves the partial plan (in the next step) with respect to the objective function.

Here are some useful observations regarding R' :

- $R - R'$ is the set of risk intervals that would be removed due to the selection of w
- $R' - R$ is the set of risk intervals that would be added due to the selection of w
- $R' \cap R$ is the set of risk intervals unaffected by w
- $R - R' = R - R' \cap R$ and $R' - R = R' - R' \cap R$ by set arithmetic

Since

$$\begin{aligned}
 \Delta(w) &= \Psi(R) && -\Psi(R') \\
 &= [\Psi(R - R \cap R') + \Psi(R \cap R')] && - [\Psi(R' - R \cap R') + \Psi(R \cap R')] \\
 &= \Psi(R - R \cap R') && -\Psi(R' - R \cap R') \\
 &= \Psi(R - R') && -\Psi(R' - R),
 \end{aligned}$$

we can calculate $\Delta(w)$ using the smaller risk sets $R - R'$ and $R' - R$. We call these the *affected risk sets*. We attach affected risk sets to each $w \in P$:

- $w.R_\Delta$: $R - R'$ if w is selected as the next planning branch
- $w.R'_\Delta$: $R' - R$ if w is selected as the next planning branch
- $w.R_v$: the subset of $w.R'_\Delta$ representing “new” risk intervals for w —those not resulting from a risk reduction to a member of $w.R_\Delta$

The specifics of how these sets are constructed and modified are left to the pseudocode below, which also makes use of the following hash tables to link each member of $w.R_\Delta$ to its changed version in $w.R'_\Delta$:

- $w.H$: the mapping from $w.R'_\Delta$ to $w.R_\Delta$, such that $w.H(r')$ is the original version of r' before risk reduction by w
- $w.H'$: the mapping from $w.R_\Delta$ to $w.R'_\Delta$, such that $w.H'(r)$ is the new version of r after risk reduction by w

We use square brackets to represent the whole image of $w.H$ or $w.H'$. Note that $w.H'[w.R_\Delta] = w.R'_\Delta - w.R_v$ and $w.H[w.R'_\Delta] = w.R_\Delta - R^-$, where R^- is the risk set removed but not changed by w (we do not store this explicit subset as we do for $w.R_v$).

In summary, note that the potential changes in R due to w result from the following possible sources:

- The new risk set for the route π of w
- The new risk set due to the fixed location of f at firing point \mathbf{c} during $[t_2, t_3]$
- The removal of a risk set and addition of a new risk set due to route replacement
- Risk reduction by w , which may include members of $w.R_v$

7. Unit Subsets

Define the *unit risk subset* as $R[e] = \{(e', f, t_a, t_b, s, \Phi) \in R \mid e' = e\}$. We assume that risk sets are indexed by units, such that we can retrieve or iterate over the members of $R[e]$ without needing to check the other members of R . Assuming that R contains the same number of risk intervals for each $e \in E$, then unit indexing reduces the cost of a risk set iteration (when only members of $R[e]$ are needed). Define

$$N_{R[e]} \approx \frac{|R|}{|E|} \quad (13)$$

as the *risk subset count* for e . Even if R has a different number of risk intervals for each unit e , $N_{R[e]}$ is probably still a good approximation for $|R[e]|$.

Similarly, for a set of fire support tasks P , define the *targeting subset* as $P[e] = \{(f, \pi, \mathbf{c}, \phi, e', t_1, t_2, t_3) \in P \mid e' = e\}$. Assuming that P contains the same number of fire support tasks targeting each $e \in E$, the *targeting subset count* for e is defined as

$$N_{P[e]} \approx \frac{|P|}{|E|} \quad (14)$$

8. Planning Branch Generation and Removal

The *planning branch generator* is any function G mapping an availability task $a = (f, \mathbf{c}, t', t'')$, a risk interval r , and a partial plan W to a set of potential fire support tasks P_a such that $\forall w = (f', \mathbf{c}' \rightarrow \mathbf{c}'', \phi, e, t_1, t_2, t_3) \in P_a : f' = f, \mathbf{c}' = \mathbf{c}, [t_1, t_3] \subseteq [t', t'']$. In other words, each new potential task $w \in G(a, r, W)$ must be for a 's unit, begin its route at a 's location, and is limited to the time interval of a . Of course, each such w must be a valid fire support task, so its route must be achievable and e must be targetable from \mathbf{c}'' given the movement and engagement capabilities of f' .

Given a potential task $w = (f, \mathbf{c} \rightarrow \mathbf{c}', \mathbf{c}'', \phi, e, t_1, t_2, t_3)$ and a set of availability tasks A , define the *generating task function* as $G'(w, A) = (f, \mathbf{c}, t', t'') \in A$ such that $[t_1, t_3] \subseteq [t', t'']$. G' gives us the availability task a that was used to generate w , i.e. $w = G(a, r, W)$. Due to the restrictions in the definition of a planning node, there should be only one such a for any $w \in P$. The following proposition, which must hold for any planning node (A, P, R, W) , relates G to G' :

$$G'(w, A) = a \leftrightarrow w \in G(a, r, W)$$

In the pseudocode below, the TryGenerateTask subroutine performs the role of G , and GeneratingTask performs the role of G' .

E. SEARCH PROBLEM CHARACTERIZATION

Since the fire support planner is an Enhancement Planner, as described in the Conceptual Planning Framework, its objective is only to maximize an objective function—in this case, that means minimizing Ψ . The two basic approaches to this are

- Optimization: what reachable planning node has the lowest cost?
- Satisfaction: does a reachable planning node with given cost M exist?

A general property of search is that optimization and satisfaction have the same upper bound complexity: if no satisficing node exists, then we end up expanding every node, just as we would for an optimization approach. However, if we can choose an acceptable threshold value M greater than the true minimum cost with high probability, then a satisficing search has a high probability of completing faster than an optimization search.

At this point, we have enough groundwork to construct a variety of search algorithms towards either approach. We make a few observations about the search space, organized by the following AI search algorithm components (Russel and Norvig 2010, ch. 3–4): goal test, successor function, cost function, and search strategy (we have already discussed the fifth, which is the definition of a search node).

1. Goal Test

A classical search problem has a goal test to determine whether a search node is a solution of the problem, but it is common for optimization problems to ignore this concept (Russel and Norvig 2010, Ch. 4). By definition, Enhancement Planners have no goal conditions, but we certainly cannot search any further from a node (A, P, R, W) if we have run out of task options $(P = \emptyset)$. Unfortunately, W in this case may not be a very good plan. If the last several branches made W worse (with respect to Ψ), then the “goal node” is farther from the optimal solution than some “non-goal” nodes. We can fix this problem by retaining a copy of the best-scoring node yet found (i.e., a single backtracking point).

Of course, if we are performing a satisficing search, a goal node is any in which $\Psi(R) \leq M$. A single algorithm can be used for either optimization or satisficing search: provide M as input, and use an impossibly low value (i.e. 0) to achieve an optimization search instead of a satisficing search. The algorithm should output the best plan found if none achieved score M .

2. Successor Function

The set P of a search node (A, P, R, W) contains the planning branches. Each $w \in P$ is a fire support task that could be added to W , along with enough information (in its affected risk sets) to calculate the difference in $\Psi(R)$ that would result. Let $w \in P$ be a choice of planning branch, and let $a_w = G'(A, w)$. The complete successor node for w is (A', P', R', W') , where

$$\begin{aligned} A' &= A - \{a_w\} \cup \text{Remainders}(a_w, w, W), \\ P' &= \bigcup_{a \in A, r \in R} G(a, r, W), \\ R' &= R - w.R_\Delta \cup w.R'_\Delta \\ W' &= \begin{cases} W \cup \{w\}, & \text{if } \varpi(W, a_w) \text{ is undefined} \\ W - \{w'\} \cup \{w, \Pi(w, w')\}, & \text{if } \varpi(W, a_w) = w' \end{cases} \end{aligned} \tag{15}$$

We have assumed a plan-space approach, meaning that the successors do not need to have any particular chronological relationship with the other members of partial plan W (except that two tasks for the same unit may not overlap in time). In particular, the search strategy is not limited to forward or backward planning.

Planning nodes contain elements with real numbers (time points and locations). Therefore, even a single non-terminal node (in which A includes at least one member of nonzero duration) has an uncountably infinite number of possible successors. No algorithm can explore this entire space; any successor function must generate new nodes using a discrete method. TryGenerateTask, the G function of the algorithm below, uses a “greedy time-maximizing” strategy: each new potential task suppresses a threat for the entire duration of a risk interval caused by that threat, or the maximal achievable subinterval. TryGenerateTask produces at most one potential task for each pair (a, r) : the one of maximum useful duration. Many other approaches are possible; that is, other versions of G could produce larger sets of options.

The expression for P' in equation (15) is a notational economy. Since many other members of P are still valid planning branches after $w \in P$ is selected, the efficient

approach to creating P' is to remove from P only the planning branches that cannot coexist with w and add only the new planning branches. The subroutine `ApplyTask`, presented below, does exactly this.

The successor function of any search algorithm determines the branching factor, which in turn affects the time complexity of the search. Since P contains all planning branches, the branching factor in our search space is $|P|$. We discuss this in more detail in the complexity analysis below, but for now, we note that P grows with A and R (which provide the inputs for G). The size of A increases each time an availability task is partitioned by the `Remainders` function, and R can grow when new fire support tasks introduce new risk intervals.

It is reasonable to assume a minimum fire support task duration t_{\min} because fire support tasks require some minimum finite amount of time—for example, the time required to fire a single shot. Potential tasks with a duration less than t_{\min} can be ignored or thrown away. This gives us an upper bound on the search depth, which we now derive.

Assume the input availability tasks are bounded by the time interval $[0, t_{\text{END}}]$ (the duration of the input plan, for example). For any search depth i , let σ_i be the total fire support effects time in the plan; that is, $\sigma_i = \sum_{(f, \pi, c, \phi, e, t_1, t_2, t_3) \in W_i} (t_3 - t_2)$ where W_i is the partial plan after step i along any search path. Similarly, let μ_i be the total fire support movement time in W_i , i.e. $\mu_i = \sum_{(f, \pi, c, \phi, e, t_1, t_2, t_3) \in W_i} (t_2 - t_1)$. Since for any fire support task $t_3 - t_2 \geq \tau_{\min}$ and $t_2 - t_1 \geq 0$, it must be that $\sigma_i \geq i\tau_{\min}$ and $\mu_i \geq 0$. The remaining availability time in A_i , $\alpha_i = \sum_{(f, c, t', t'') \in A_i} (t'' - t')$, is the original availability time minus the time used by all fire support tasks in W_i : $\alpha_i = \alpha_0 - (\sigma_i + \mu_i)$. The maximum availability time per unit $f \in F$ before the first iteration is $t_{\text{END}} - 0$, so $\alpha_0 \leq |F|t_{\text{END}}$. Therefore, even at the greatest possible value of α_0 and minimum possible value of $\sigma_i + \mu_i$, $\alpha_i = |F|t_{\text{END}} - i\tau_{\min}$ and $\alpha_i = 0$ when i reaches the following value:

$$i_{\max} = \left\lceil F \right\rceil \frac{t_{\text{END}}}{\tau_{\min}} \quad (16)$$

No more tasks can be assigned when no availability time remains, so i_{\max} is the maximum search depth.

3. Plan Cost

As mentioned above, the objective function $\Psi(R)$ serves as the cost of a planning node. This choice has an impact on the search strategy (see the next section) because $\Psi(R)$ is not monotonic, as we now demonstrate with an example.

In the situation shown in Figure 27, the input plan consists of only Friendly Unit 1 advancing and assaulting Enemy Unit 1. It is threatened by Enemy Units 1 and 2. Friendly Units 2 and 3 are the only available fire support units, and they have not yet been assigned any fire support tasks. The dotted lines (routes) and question marks (firing positions) indicate potential fire support tasks; the planner must choose one of them along with a target as its next fire support task. The X marks a position that is too far away to engage Enemy Unit 1. Unfortunately, all of these tasks result in a greater risk value for the resulting partial plan because they introduce new risk intervals (mainly from Enemy Unit 2) with risk values that outweigh the reduction of risk to Friendly Unit 1. This represents a local minimum in the plan search.

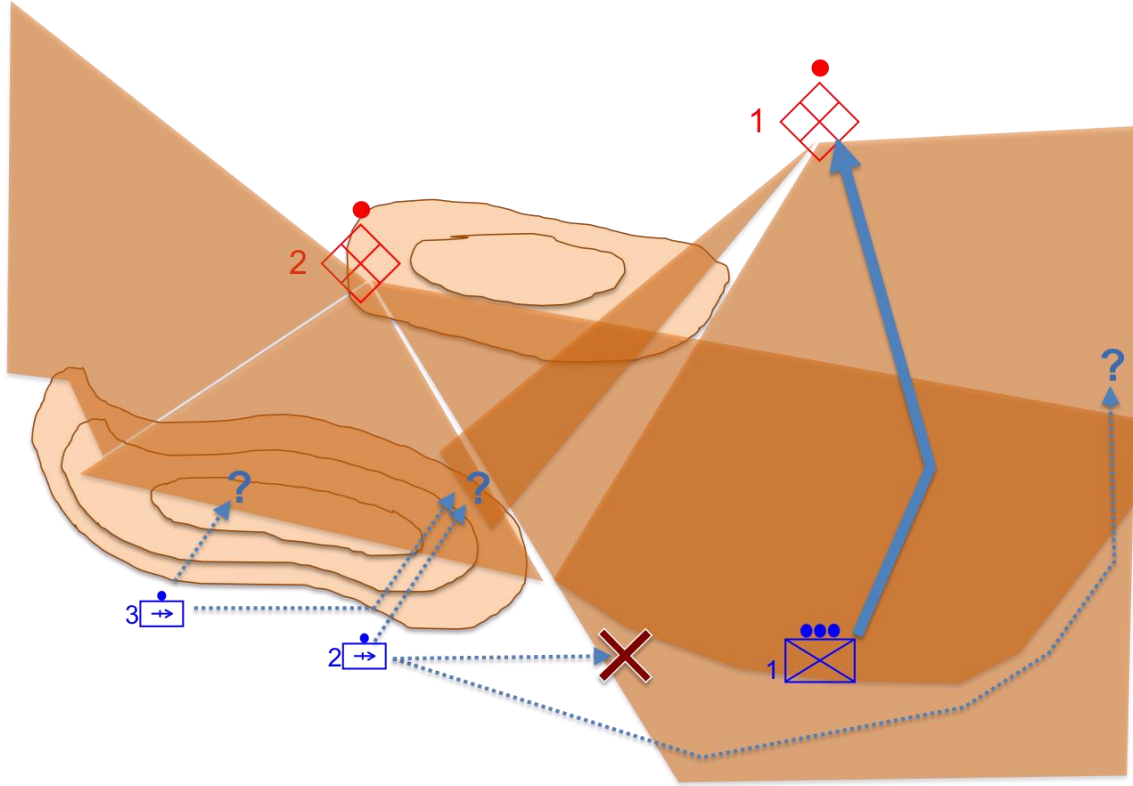


Figure 27. Increasing and Decreasing Risk Value, Part 1

A solution to this problem is to use both Friendly Units 2 and 3 to suppress the two enemy units simultaneously, as shown in Figure 28. This decreases the residual risk not only to Friendly Unit 1, but to Friendly Units 2 and 3 as well. This is called *mutual support* and is a basic military tactical concept. We would expect the risk value of this plan to be less than that of the input plan. Adding one task at a time, there is no way to reach the better plan without first passing through a worse one, according to the Ψ score. This is similar to the concept of interfering effects, found in many automated planning domains (Ghallab, Nau, and Traverso 2004, 321–323). The good news is that the “interference” here is beneficial from the attacker’s point of view. Our algorithm deals with the “worse before better” problem with a single backtracking point, as already mentioned in the discussion of the goal test.

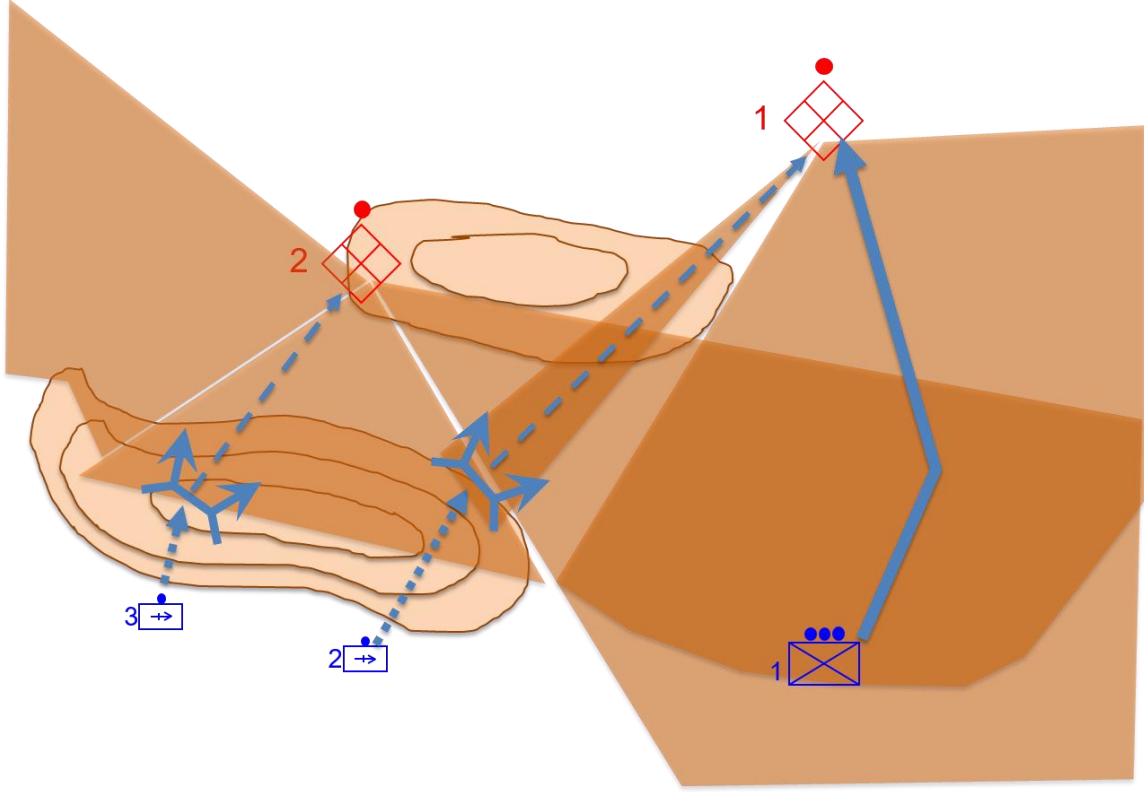


Figure 28. Increasing and Decreasing Plan Risk Value, Part 2

Note that Ψ is a derived model, not an oracle. It is entirely possible that some plan W , with a better Ψ score than another plan W' , actually results in a worse measure of effectiveness under multiple replications in the CSE—even if that measure is actual losses to the Friendly Force (the natural run-time equivalent to the “expected losses” of Ψ). This is not a problem as long as Ψ does reasonably well in generating useful fire support plans. After all, human planners cannot predict which plan will result in a better measure of effectiveness, either—if they could, they would not need to run simulation-based experiments. We test our implementation against measures of effectiveness in Chapter VI.

4. Search Strategy

Our search strategy is Greedy Best-First Search (Doran and Michie 1966), where we interpret the task score $\Delta(w)$ as the heuristic (recall that Δ is just the difference in

Ψ score). This means that our algorithm follows only one search path, depending on the planning branch generator to provide good tasks and Δ to identify them.

We could not have selected uniform cost search or A* search due to the increasing/decreasing nature of Ψ . There are many other search strategies to choose from for a cost function of this nature, but Greedy Best-First Search is the fastest. It cannot guarantee optimality, and its only protection against local minima is the single backtracking point. However, planning problems with spatial and temporal dimensions tend to have massive search spaces, even when discretized. Greedy Best-First Search avoids some of the combinatorial quagmire with its non-exhaustive strategy. The rather large number of planning branches (in P) helps counterbalance the potential shortcomings of the strategy.

F. ALGORITHM

We now present our Greedy Best-First Search algorithm for fire support planning. Let W_0 be the input plan. We use the following variables in computational complexity arguments following each subroutine:

- $n = |E| + |F|$: the number of units
- $m = \frac{t_{\text{END}}}{\tau_{\text{min}}}$: the maximum number of fire support tasks per unit (see Equation (16))
- $v = |U|$: the number of terrain nodes
- k_x : an upper bound on the number of actions per route, including the routes in W_0 and any route for a fire support task
- k_v : an upper bound on $|w.R_v|$ for any possible fire support task w

Each running time expression $T_i(\bullet)$ and, in some cases, counting expression $N_i(\bullet)$, corresponds to Algorithm i and is defined in subsection i . We refer to some of these before they are defined because a top-down description of the algorithm leads to better understanding.

1. Top-Level Procedure

Algorithm 1 is the entry point to the plan search. It takes as input a partial plan W , a set of availability tasks A , and a maximum desired plan cost M , and it returns a new version of W with fire support tasks assigned to available units. If the algorithm discovers a plan with an objective function score less than or equal to M , it immediately returns that plan. Otherwise, it keeps adding fire support tasks until running out of planning branches and then returns the best-scoring plan W_α that it found.

Algorithm 1 PlanFireSupport (W : maneuver plan, A : availability task set, M : maximum cost)

```

 $W_\alpha \leftarrow W$ 
 $R \leftarrow \text{BuildRiskIntervals}(W)$ 
 $\alpha \leftarrow \Psi(R)$ 
for all  $a \in A$  do
     $P \leftarrow \text{PUGenerateTasks}(a, R, R, W)$ 
while  $\Psi(R) > M$  and  $P \neq \emptyset$  do
     $w \leftarrow \text{ChooseTask}(P)$ 
     $(A, P, R, W) \leftarrow \text{ApplyTask}(A, P, R, W, w)$ 
    if  $\Psi(R) < \alpha$  then
         $\alpha \leftarrow \Psi(R)$ 
         $W_\alpha \leftarrow W$ 
return  $W_\alpha$ 

```

We call the two major steps of PlanFireSupport *initialization* and the *planning loop*, the latter of which is everything inside the **while** block. Let $k_{Ai}, k_{Pi}, k_{Ri}, k_{Wi}$ be the sizes of the planning node components (that is, $|A|, |P|, |R|, |W|$) at the beginning of iteration i of the planning loop (starting from $i=1$). Let k_{zi} be an upper bound on the number of risk segments contained in any single member of R at the beginning of iteration i . Note the following:

- $k_{W1} = |W_0|$
- After R is constructed by BuildRiskIntervals, $k_{R1} = N_B(|E|, |W_0|, k_x) = O(nk_{W1}k_x)$ by Equation (6)

Initialization proceeds as follows:

- Copy the input plan into W_α as the default best plan: $O(k_{w_1}k_x)$ time
- BuildRiskIntervals: $T_B(|E|, |W_0|, k) = O(nk_{w_1}k_x T_\Psi)$ time (Equation (5))
- Calculation of $\Psi(R)$: $O(k_{R_1}) = O(nk_{w_1}k_x)$ time (Equation (8))
- Due to the **for** loop, we call GenerateTasks against each of the initial availability tasks to create the initial set P in $O(k_{A_1})T_4(k_{R_1}, k_{R_1}, k_{w_1}, k_{z_1})$ time

The planning loop contains the simple logic of Greedy Best-First Search: choose the best-scoring task in P , based only on the single-step improvement to the objective function Ψ , and use it to create the next planning node. We also update the best plan W_α and best plan score α if needed. If we ever manage to generate a plan with cost M or less, we return that plan immediately. The maximum number of iterations is mn (Equation (16)). Each iteration has the following steps:

- ChooseTask: $T_2(k_{p_i})$ time
- ApplyTask: $T_3(k_{A_i}, k_{p_i}, k_{R_i}, k_{w_i}, k_{z_i})$ time
- Computation of $\Psi(R)$: $T_{\Psi(R)} = O(k_{R_i})$ time
- Copying W to W_α : $O(k_{w_i}k_{z_i})$ time

The initialization plus planning loop time is the total running time of the algorithm:

$$T_1 = O(k_{w_1}k_x + nk_{w_1}k_x T_\Psi + nk_{w_1}k_x + k_{A_1}T_4(k_{R_1}, k_{R_1}, k_{w_1}, k_{z_1})) \\ + \sum_{i=1}^{mn} [T_2(k_{p_i}) + T_3(k_{A_i}, k_{p_i}, k_{R_i}, k_{w_i}, k_{z_i}) + O(k_{R_i}) + O(k_{w_i}k_{z_i})],$$

which simplifies to

$$T_1 = O(nk_{w_1}k_x T_\Psi + k_{A_1}T_4(k_{R_1}, k_{R_1}, k_{w_1}, k_{z_1})) \\ + \sum_{i=1}^{mn} [T_2(k_{p_i}) + T_3(k_{A_i}, k_{p_i}, k_{R_i}, k_{w_i}, k_{z_i}) + O(k_{R_i}) + O(k_{w_i}k_{z_i})] \quad (17)$$

2. Choosing the Best Task

Algorithm 2 selects a planning branch. We use a special heuristic here: always choose a task with a null route when one exists in P . In other words, fire support units never change firing positions unless they have to. Apart from this consideration, we choose the planning branch with the best score $\Delta(w)$.

Algorithm 2 ChooseTask(P : potential task set)

```

 $N \leftarrow \{p \in P \mid p \text{ has a null route}\}$ 
if  $N \neq \emptyset$  then
     $P' \leftarrow N$ 
else
     $P' \leftarrow P$ 
return  $\operatorname{argmax}_{w \in P'} \Delta(w)$ 

```

Assuming we maintain the set N of potential tasks with null routes along with the larger set P , the first five lines take only constant time. The argmax can be achieved with an exhaustive linear search of the stored values $\Delta(w)$ for each $w \in P'$ (a faster search does not improve the overall time complexity of the planner). Since $P' \subseteq P$, this search requires no more than $k_p = |P|$ steps, and

$$T_2(k_p) = O(k_p) \quad (18)$$

3. Applying the Selected Task

Once a planning branch w has been selected, Algorithm 3 generates the new planning node. This is our realization of Equation (15).

Algorithm 3 ApplyTask(A : availability set, P : potential task set, R : risk set, W : partial plan, $w = (f, \pi, \mathbf{c}, e, t_1, t_2, t_3)$: fire support task)

```

 $a_w \leftarrow \text{GeneratingTask}(w, A)$ 
 $\{a', a''\} \leftarrow \text{Remainders}(a_w, w, W)$ 
 $A \leftarrow A - \{a_w\} \cup \{a', a''\}$ 
 $P \leftarrow P - \text{GenerateTasks}(a_w, R, R, W)$ 
for all  $p \in P[e]$  do
    UpdateAffectedRiskIntervals( $p, w, w.R'_\Delta[e], w.R_\Delta$ )
for all  $p = (f', \pi', \mathbf{c}', e', t'_1, t'_2, t'_3) \in P - P[e]$  do
    UpdateAffectedRiskIntervals( $p, w, w.R_\nu[e'], w.R_\Delta$ )
 $W \leftarrow W \cup \{w\}$ 
if  $\varpi(W, a_w)$  is defined then
     $w' \leftarrow \varpi(W, a_w)$ 
     $w'' \leftarrow \Pi(w, w')$ 
     $W \leftarrow W - \{w'\} \cup \{w''\}$ 
 $R \leftarrow R - w.R_\Delta \cup w.R'_\Delta - w.R_\nu$ 
 $P \leftarrow P \cup \text{GenerateTasks}(a', R, R, W) \cup \text{GenerateTasks}(a'', R, R, W)$ 
 $R \leftarrow R \cup w.R_\nu$ 
for all  $a \in A$  do
     $P \leftarrow P \cup \text{GenerateTasks}(a, R, w.R_\nu, W)$ 
return  $(A, P, R, W)$ 

```

Let k_A, k_P, k_R, k_W be the sizes of the input sets A, P, R, W . Let k_z be an upper bound on the number of risk segments contained in any single member of R . Let k_Δ be an upper bound on the size of the affected risk set of w (i.e. $w.R_\Delta \cup w.R'_\Delta$) or any member of P . Since any such w can only reduce the risk caused by a single $e \in E$, and $N_{R[e]} \approx \frac{|R|}{|E|}$ (Equation (13)), we assume that $k_\Delta = O\left(\frac{k_R}{n} + k_\nu\right)$. The k_ν new risk intervals for w are not necessarily limited to e .

First, the availability task $a_w \in A$ that was used to generate w is replaced with its Remainders, increasing the size of A by one. Then, w and all other tasks generated from a_w are removed from P , reducing its size by $N_4(k_R)$. Assuming we maintain mappings between members of A and P and structure P by subsets (one for each member of A), the first four lines can be accomplished in constant time. Note that GenerateTasks in the

fourth line is considered a lookup of potential tasks that were already created rather than a function call to create them.

UpdateAffectedRiskIntervals changes the affected risk sets of the remaining members of P to account for the selection of w , a process described in its own subroutine (below). The first **for** loop handles all members of P targeting the same threat (e) as w . Assuming P contains an equal distribution of fire support tasks against the members of E ,¹⁸ we can use Equation (14) to say that this loop is limited to $O\left(\frac{k_p}{n}\right)$ iterations.

$|w.R'_\Delta| = O(k_\Delta) = O\left(\frac{k_R}{n} + k_v\right)$, so $|w.R'_\Delta[e]| = O\left(\frac{k_R + k_v}{n}\right)$. Since $|w.R_\Delta|$ and $|p.R_\Delta \cup p.R'_\Delta|$ are both $O(k_\Delta)$, the running time of the first loop is $O\left(\frac{k_p}{n} T_6\left(\frac{k_R + k_v}{n}, k_\Delta, k_\Delta, k_z\right)\right)$.

The second **for** loop handles the remaining members of P . None of these fire support tasks can affect any member of $w.R'_\Delta - w.R_v$ because they target a different unit than e , so we only pass $w.R_v[e']$ to UpdateAffectedRiskIntervals. $|w.R_v[e']| = O\left(\frac{k_v}{n}\right)$, so the second loop's running time is $O\left(k_p T_6\left(\frac{k_v}{n}, k_\Delta, k_\Delta, k_z\right)\right)$.

Next, w is added to W in constant time. Route replacement is needed only if $\varpi(W, a_w)$ is defined. If so, we replace the immediately following task w' with the task $\Pi(w, w')$. Since $\Pi(w, w')$ must have been created already—when w was originally added to P —we assume that it can be retrieved in constant time.

R is then updated using the affected risk sets of w . We do not add $w.R_v$ yet in order to avoid duplicate tasks in the next line. Assuming the members of $w.R_\Delta$ are references to the appropriate members of R , the set subtraction takes only $|w.R_\Delta|$ steps. The addition of $w.R'_\Delta$ (less $w.R_v$) takes $O(|w.R'_\Delta|)$ steps. The entire line requires $O(k_\Delta)$.

¹⁸ See GenerateTask, below, for the support to this assumption.

The upper-bound size of R (i.e. k_R) has not increased at this point because only the *changed* risk intervals have been added; each $r \in w.R'_\Delta - w.R_\nu$ has a corresponding member $w.H(r) \in w.R_\Delta$ that was simultaneously removed.

GenerateTasks is called for the two new members of A against all risk intervals in R , taking $2T_4(k_R, k_R, k_W + 1, k_z)$ steps. This increases the size of P by $2N_4(k_R)$, for a net gain (at this point) of $N_4(k_R)$. Afterwards, $w.R_\nu$ can be safely added to R , requiring k_ν steps and increasing its upper bound size to $k_R + k_\nu$. The final steps, contained in the last **for** loop, are to call GenerateTasks for each of the $k_A + 1$ availability tasks against each of the k_ν new risk intervals. This takes $O((k_A + 1)T_4(k_R, k_\nu, k_W + 1, k_z))$ time and increases the size of P by $O((k_A + 1)N_4(k_\nu))$.

The total running time of ApplyTask is

$$\begin{aligned}
T_3(k_A, k_P, k_R, k_W, k_z) = & \\
& O\left(\frac{k_P}{n}T_6\left(\frac{k_R + k_\nu}{n}, k_\Delta, k_\Delta, k_z\right) + k_P T_6\left(\frac{k_\nu}{n}, k_\Delta, k_\Delta, k_z\right)\right) + \\
& O(k_\Delta + 2T_4(k_R, k_R, k_W + 1, k_z) + (k_A + 1)T_4(k_R, k_\nu, k_W + 1, k_z)), \\
& \text{where } k_\Delta = \frac{k_R}{n} + k_\nu
\end{aligned} \tag{19}$$

The sizes of the updated sets A , P , R , W are

$$|A| = k_A + 1 \tag{20}$$

$$|P| = k_P + N_4(k_R) + O((k_A + 1)N_4(k_\nu)) \tag{21}$$

$$|R| = O(k_R + k_\nu) \tag{22}$$

$$|W| = k_W + 1 \tag{23}$$

4. Generating Potential Fire Support Tasks

GenerateTasks accepts an availability task a , a risk set R , and a partial plan W as input. Conceptually, it returns the set $P' = \bigcup_{r \in R} G(a, r, W)$, where each $p \in P'$ has the correct affected risk sets attached.

Algorithm 4 GenerateTasks($a = (f, \mathbf{c}, t_0, t_4)$: availability task, R, R' : risk sets, W : partial plan)

```

 $E' \leftarrow$  the set of all enemy units appearing in  $R'$ 
 $Q \leftarrow \text{new PositionSearchQueue}(f, \mathbf{c}, E')$ 
 $P' \leftarrow \emptyset$ 
while  $Q$  is not empty do
     $(\pi, \mathbf{c}', e) \leftarrow Q.\text{dequeue}$ 
    for all  $r \in R'[e]$  do
         $P' \leftarrow P' \cup \text{TryGenerateTask}(a, r, \pi, \mathbf{c}', R, W)$ 
return  $P'$ 

```

Let $k_R = |R|$, $k'_R = |R'|$, $k_W = |W|$. Let k_z be an upper bound on the number of risk segments contained in any single member of R . Note that this R is not always the entire risk set of the partial plan, so E' is not always the entire Enemy Force—however, for complexity analysis we use n as an upper bound for $|E'|$. On the first line, we can iterate through R' in k'_R steps to construct E' . The PositionSearchQueue Q handles pathfinding for new potential tasks and is described in the Pathfinding section below. It requires time $T_8(v)$ to generate $N_8(n)$ path tuples.

The **while** loop iterates through every path tuple (π, \mathbf{c}', e) in Q , where π is a route, \mathbf{c}' is a firing position, and $e \in E'$. We invoke TryGenerateTask on every $r \in R'$ matching the same e . Each call produces up to N_5 new fire support tasks for P' in time $T_5(k_R, k_W)$. The inner **for** loop examines $N_{R[e]} = O\left(\frac{k'_R}{n}\right)$ risk intervals (Equation (13)),

and the **while** loop has $N_8(n)$ iterations, so the total running time of the **while** loop is $O(N_8(n)k'_R T_5(k_R, k_W, k_z)/n)$.

The total time for GenerateTasks is

$$T_4(k_R, k'_R, k_W, k_z) = O(k'_R + T_8(v) + N_8(n)k'_R T_5(k_R, k_W, k_z)/n) \quad (24)$$

The maximum number of new fire support tasks that can be created is N_5 per iteration, for a total of

$$N_4(k'_R) = O(N_8(n)k'_R N_5 / n) \quad (25)$$

5. Generating a Task Option for a Single Risk Interval

TryGenerateTask accepts an availability task a , a risk interval r , a route π , a firing position \mathbf{c}' , the partial plan's risk set R , and the partial plan W as input. It produces a set with up to one element: a potential fire support task using route π to reach position \mathbf{c}' , and from there to engage Enemy Force unit e (taken from r), for the maximum achievable subinterval of r , constrained by the time interval of a and the time required to traverse π . If this turns out to be unachievable within the time limits of a , it returns the empty set. Before returning the new potential task (if successful in creating one), it attaches the affected risk sets, accounting for route replacement if needed.

Algorithm 5 TryGenerateTask($a = (f, \mathbf{c}, t_0, t_4)$: availability task, $r = (e, f, t_a, t_b, s)$: risk interval, π : route, \mathbf{c}' : position, R : risk set, W : partial plan)

```

 $\tau \leftarrow$  time required for  $f$  to transit  $\pi$ 
if  $\varpi(W, a)$  is undefined then
     $t' \leftarrow t_4$ 
else
     $w' \leftarrow (f, \pi', \mathbf{c}'', e', t_4, t_5, t_6) \leftarrow \varpi(W, a)$ 
     $\pi'' \leftarrow$  the route of  $\Pi(w, w')$ 
     $\tau'' \leftarrow$  time required for  $f$  to transit  $\pi''$ 
     $t' \leftarrow t_5 - \tau''$ 
 $t_3 \leftarrow \min(t', t_b)$ 
 $t_2 \leftarrow \max(t_0 + \tau, t_a)$ 
 $t_1 \leftarrow t_2 - \tau$ 
if  $t_0 \leq t_1 \leq t_2 \leq t_3 \leq t'$  and  $t_3 - t_2 \geq \tau_{min}$  then
     $w \leftarrow (f, \pi, \mathbf{c}', e, t_1, t_2, t_3)$ 
     $R^+ \leftarrow \text{BuildRiskIntervals}(\{w\})$ 
    if  $\varpi(W, a)$  is defined then
         $R^- \leftarrow w'.R_\nu$ 
         $R^+ \leftarrow R^+ \cup \text{BuildRiskIntervals}(\{\Pi(w, w')\})$ 
        StoreAffectedRiskIntervals( $w, R, R^+, R^-, W$ )
    return  $\{w\}$ 
else
    return  $\emptyset$ 

```

The temporary variables τ, τ'' are the amount of time required for f to traverse the route π for the new task and the route π'' of the route replacement task (if applicable), respectively. The first step is to determine t' , the maximum time point usable by the new task, which may be less or greater than t_4 if an immediately following task exists. The new task's end time t_3 is set to either t' or the end time of the targeted risk interval, t_b . Similarly, the effects time t_2 is set to either the earliest achievable start time $t_0 + \tau$ or the start time of the targeted risk interval t_a , whichever is later. The start time t_1 of the new task—the time f will begin traversing π —is computed such that f will arrive “just in time” at \mathbf{c}' , leaving the maximum amount of untasked time prior to t_1 for other tasks.

If the task is unachievable in the given time, then the string of inequalities $t_0 \leq \dots \leq t'$ will not hold. We also refuse to create the task if the suppression time $t_3 - t_2$ is less than minimum useful duration τ_{\min} .

If w passes muster, we use temporary risk sets R^+ and R^- to help construct the affected risk sets. R^+ is initialized to the new risk set for w , generated by the BuildRiskIntervals subroutine. Its initial size is $N_B(|E|, 1, k_x) = O(nk_x)$ by Equation (6). If route replacement is needed, then R^- is the set of $O(nk_x)$ new risk intervals that w' already added to R —which will need to be replaced if w' is replaced by $\Pi(w, w')$. We add the $O(nk_x)$ new risk intervals for $\Pi(w, w')$ to R^+ as well, which does not change its asymptotic size. At this point, we have enough information to determine the affected risk intervals for w , a task handled by StoreAffectedRiskIntervals.

Let $k_R = |R|$, $k_W = |W|$. Let k_z be an upper bound on the number of risk segments contained in any single member of R . Most of the steps in this subroutine are constant-time arithmetic or conditional checks. Here are the steps that have a more significant running time:

- Generation of $\Pi(w, w')$: requires pathfinding for the new route in time $T_8(v)$
- Two BuildRiskIntervals calls, each on a single fire support task whose route contains $O(k_x)$ actions: $2T_B(n, 1, k_x) = O(nk_x T_\Psi)$ time by Equation (5)
- StoreAffectedRiskIntervals: $O(T_7(k_R, nk_x, nk_x, k_W, k_z))$

The total running time of TryGenerateTask is

$$T_5(k_R, k_W, k_z) = T_8(v) + O(nk_x T_\Psi) + T_7(k_R, nk_x, nk_x, k_W, k_z) \quad (26)$$

The maximum number of fire support tasks that it can generate is 1, so

$$N_5 = O(1) \quad (27)$$

The maximum number of risk segments in any member of the affected risk sets of w depends on `StoreAffectedRiskIntervals`, which is described in the next section.

6. Updating Affected Risk Sets

`UpdateAffectedRiskIntervals` takes a potential fire support task p and another fire support task w (which is either part of the partial plan or about to be added to it) and updates the affected risk sets and score of p to account for the existence of w . Rather than use the attached risk sets $w.R'_\Delta$ and $w.R_\Delta$, new and old risk sets are passed explicitly in the parameters R^+ and R^- (the reason for this is explained in the next section). Since R^+ contains the new risk intervals from the perspective of p , R^+ should be disjoint with $p.R_\nu$.

Algorithm 6 `UpdateAffectedRiskIntervals`($p = (f, \pi, c, e, t_1, t_2, t_3), w = (f_w, \pi_w, c_w, e_w, t_{w1}, t_{w2}, t_{w3})$: fire support tasks, R^+, R^- : risk sets)

Require: $R^+ \cap p.R_\nu = \emptyset$
 $p.R_\Delta \leftarrow p.R_\Delta - R^-$
 $p.R'_\Delta \leftarrow p.R'_\Delta - p.H'[R^-]$
for all $r_w \in R^+$ **do**
 $r_{wp} \leftarrow \lambda(r_w, p)$
 if $r_w \neq r_{wp}$ **then**
 $p.R_\Delta \leftarrow p.R_\Delta \cup \{r_w\}$
 $p.R'_\Delta \leftarrow p.R'_\Delta \cup \{r_{wp}\}$
 $p.H'(r_w) \leftarrow r_{wp}$
 $p.H(r_{wp}) \leftarrow r_w$
for all $r_\nu \in p.R_\nu[e_w]$ **do**
 $r_{\nu w} \leftarrow \lambda(r_\nu, w)$
 if $r_\nu \neq r_{\nu w}$ **then**
 $p.R_\nu \leftarrow p.R_\nu - \{r_\nu\}$
 $p.R_\nu \leftarrow p.R_\nu \cup \{r_{\nu w}\}$
 $p.R'_\Delta \leftarrow p.R'_\Delta \cup \{r_{\nu w}\}$
 $\Delta(p) \leftarrow \Psi(p.R_\Delta) - \Psi(p.R'_\Delta)$

Let $k_R^+ = |R^+|$, $k_R^- = |R^-|$, $k_\Delta = |p.R_\Delta \cup p.R'_\Delta|$. Let k_z be an upper bound on the number of risk segments contained in any single member of $p.R_\Delta$, $p.R'_\Delta$, or R^+ . The first step is to remove R^- from $p.R_\Delta$. These risk intervals have already been removed from

the current planning node by w , so p has no reason to keep track of them; likewise for p 's modified versions $p.H'[R^-]$. Assuming each affected risk set is implemented as a hash table, we can accomplish the set subtraction in $2k_R^-$ steps.

The first **for** loop accounts for the new risk intervals in R^+ . We use p to reduce the risk of each of these in $T_\lambda(k_z) = O(k_z T_\Psi)$ time (Equation (11)). If this results in a different risk interval, then we add it to the affected risk set in constant time. The total running time of the first loop is $O(k_R^+ k_z T_\Psi)$.

The second **for** loop does the reverse operation. Since the members of $p.R_\nu$ will not be added to the partial plan's risk set until (and if) p is selected as a planning branch, w has never "seen" them. We use w to reduce the risk of each of these in $O(k_z T_\Psi)$ time. There is no need to modify $p.R_\Delta$ or the hash mappings because nothing in $p.R_\nu$ exists in the current partial plan. The running time of the second loop is $O(k_\nu k_z T_\Psi)$.

The final step is to store the updated task score for p . Using Equation (8), we need $T_{\Psi(p.R_\Delta)} + T_{\Psi(p.R'_\Delta)} = O(k_\Delta)$ time for the calculation.

The total running time of UpdateAffectedRiskIntervals is

$$\begin{aligned} T_6(k_R^+, k_R^-, k_\Delta, k_z) &= O(k_R^- + k_R^+ k_z T_\Psi + k_\nu k_z T_\Psi + k_\Delta) \\ &= O(k_R^- + (k_R^+ + k_\nu) k_z T_\Psi + k_\Delta) \end{aligned} \quad (28)$$

Since $R^+ \cap p.R_\nu = \emptyset$, risk reduction is only performed on each member of R^+ and $p.R_\nu$ once. According to Equation (12), the new members of $p.R_\Delta \cup p.R'_\Delta$, may have no more than 2 more risk segments than those of R^+ or $p.R_\nu$. At the end of the subroutine, the maximum number of risk segments in any member of $p.R_\Delta \cup p.R'_\Delta$ is

$$N_6(k_z) = k_z + 2 \quad (29)$$

7. Storing Affected Risk Sets

The `StoreAffectedRiskIntervals` subroutine takes a fire support task w , the risk set R of the current partial plan, risk sets R^+ (disjoint with R) and R^- (a subset of R), and partial plan W . It attaches modified subsets of these risk sets that agree with the definitions of $w.R_\Delta$, $w.R'_\Delta$, and $w.R_\nu$ given above (see Task Scores and Affected Risk Sets), and it creates the hash maps $w.H$ and $w.H'$. Finally, it places the score for w in the Δ table.

Algorithm 7 `StoreAffectedRiskIntervals`($w = (f, \pi, \mathbf{c}, e, t_1, t_2, t_3)$: fire support task, R, R^+, R^- : risk sets, W : partial plan)

Require: $R^+ \cap R = \emptyset, R^- \subseteq R$

$w.R'_\Delta \leftarrow w.R_\nu \leftarrow \{\lambda(r, w) \mid r \in R^+\}$

for all $w' \in W$ **do**

`UpdateAffectedRiskIntervals`($w, w', \emptyset, \emptyset$)

$w.R_\Delta \leftarrow R^-$

for all $r \in R[e] - R^-$ **do**

$r_w \leftarrow \lambda(r, w)$

if $r \neq r_w$ **then**

$w.R_\Delta \leftarrow w.R_\Delta \cup \{r\}$

$w.R'_\Delta \leftarrow w.R'_\Delta \cup \{r_w\}$

$w.H'(r) \leftarrow r_w$

$w.H(r_w) \leftarrow r$

$\Delta(p) \leftarrow \Psi(p.R_\Delta) - \Psi(p.R'_\Delta)$

Let $k_R = |R|, k_R^+ = |R^+|, k_R^- = |R^-|, k_W = |W|$. Let k_z be an upper bound on the number of risk segments contained in any single member of R , and let k'_z be an upper bound on the number of risk segments contained in any member of $w.R_\Delta \cup w.R'_\Delta$. We will update an expression for k'_z as we proceed through the following arguments.

R^+ contains the new risk intervals for w before any risk reduction has been performed on them—which implies that its members contain $O(k_x)$ risk segments. The first line of the subroutine (after the **Require** line) performs risk reduction on each member of R^+ by w itself, requiring $k_R^+ T_\lambda(O(k_x)) = O(k_R^+ k_x T_\Psi)$ time (Equation (11)).

We then store the resulting set in $w.R_v$ and $w.R'_\Delta$ (recall that $w.R_v \subseteq w.R'_\Delta$). $w.R_v$ now represents the new risk intervals of w as if w were the only task in the partial plan. Since only one risk reduction has been performed on each member of $w.R'_\Delta$ and each risk reduction increases the number of risk segments per risk interval by at most 2 (Equation (12)), $k'_z \geq k_x + 2$.

To account for the k_w members of W , we then call `UpdateAffectedRiskIntervals` on each of them. These invocations use empty sets for the parameters R^+, R^- (because the additions and removals due to each w' are already accounted for in R , which will be handled in the next loop). Let k'_{zi} be an upper bound on the number of risk segments in any member of $w.R_v$ in iteration i of this **for** loop. Based on the arguments from the last paragraph, $k'_{z1} = k_x + 2$. Since $N_6(k'_{zi}) = k'_{zi} + 2$ (Equation (29)), we can update our risk segment count to $k'_z \geq k'_{z,k_w} = k_x + 2 + 2k_w$. This loop requires $O(k_w T_6(0, 0, k_v, k_z))$ time.

The members of R^- came from R and will disappear if w becomes part of the plan, so we do not need to perform risk reduction on them. We simply use them as the initial members of $w.R_\Delta$. We can copy by reference in $O(k_R^-)$ time. Since the members of R^- have up to k_z risk segments, we now have $k'_z \geq \max(k_x + 2 + 2k_w, k_z)$.

Next, we need to determine which members of R would be affected by w . We only need to consider the subset $R[e] - R^-$ because w only targets e and, as we just mentioned, R^- will be completely removed from R if w is selected. (Although R^- occurs “after” w chronologically, it is possible for w to overlap R^- in time if the replaced route takes less time to traverse than the replacing route. This would result in the illogical case of a unit firing from a fixed position in support of its own movement.) We consider each r individually, rather than with set-builder notation, because we need to create the mappings $w.H' = r$ and $w.H = r_w$ in the event that w affects r . Using Equation (13),

$$|R[e] - R^-| = \frac{k_R - k_R^-}{n}. \text{ Inside this } \mathbf{for} \text{ loop, the first line has running time}$$

$T_\lambda(k_z) = O(k_z T_\Psi)$, so the whole loop takes $O\left(k_z T_\Psi \left(\frac{k_R - k_R^-}{n}\right)\right)$. Any risk interval r added to $w.R_\Delta$ is a member of R , and therefore has no more than k_z risk segments. Any risk interval r_w added to $w.R'_\Delta$ is the result of just one risk reduction $(\lambda(r, w))$ to a member of R , so we now must say that $k'_z \geq \max(k_x + 2 + 2k_w, k_z + 2) = \max(k_x + 2k_w, k_z) + 2$.

The last line just stores the task score in Δ . The risk calculations have running time $T_{\Psi(w.R_\Delta)} + T_{\Psi(w.R'_\Delta)} = O(|w.R_\Delta| + |w.R'_\Delta|)$, but these two sets were constructed by copying operations earlier in the subroutine, so the running time of the calculation is of the same order as the earlier steps. The total running time of StoreAffectedRiskIntervals is

$$T_7(k_R, k_R^+, k_R^-, k_w, k_z) = O\left(k_R^+ k_x T_\Psi + k_w T_6(0, 0, k_v, k_z) + k_R^- + k_z T_\Psi\left(\frac{k_R - k_R^-}{n}\right)\right) \quad (30)$$

Based on the upper bound arguments for k'_z , the maximum number of risk segments in an affected risk interval of w at the end of this subroutine is

$$N_7(k_z) = \max(k_x + 2k_w, k_z) + 2 \quad (31)$$

8. Pathfinding

The fire support planner relies on a Pathfinding subsystem, as described in the Conceptual Planning Framework. An approach based on the A* tactical pathfinding algorithm (van der Sterren 2002) is a reasonable assumption (and the choice for our implementation, described in Chapter VI), but the algorithm is loosely coupled with the Pathfinding capabilities. The only requirement is that Pathfinding can perform *point-to-point* and *multi-target* searches, which we define shortly. Theoretically, any Pathfinding system that can perform these types of searches can be used to support the fire support planner, but the quality of resulting plans is obviously dependent upon the quality of the Pathfinding results. We assume here, for simplicity, that the nodes of the terrain are the pathfinding search nodes.

Point-to-point, or “conventional” pathfinding, is the case where the input is a single start location and a single end location, along with a derived movement model to provide information such as travel speed and traversable terrain types. The output is a single route. The fire support planner only uses point-to-point pathfinding for route replacement, since in that case we have the exact starting location (the firing point of a potential fire support task) and ending location (the firing point of the immediately following task).

Multi-target pathfinding uses a known start location, but the goal is to find *several* paths to *several* end locations with certain features. The end locations are not known at the start of the search, but are discovered by a special *goal node test* that checks whether the current search node contains a useful firing position (we will make this more concrete). The g -cost function for point-to-point pathfinding may be used as the g -cost for multi-target pathfinding. The typical h -cost function for point-to-point pathfinding, Euclidean distance, cannot be used for multi-target pathfinding because we do not have an endpoint to which we could measure a distance. We assume a constant h -cost of 0 for multi-target pathfinding, which makes it equivalent to uniform-cost search.

An additional feature of multi-target pathfinding is that it continues searching after a goal node is found. This avoids the inefficiency of a series of point-to-point searches, which would end up exploring many of the same nodes with each restart—an especially troubling issue for tactical pathfinding implementations in which routes often explore many nodes looking for safer, rather than faster, routes. The *search goal test* determines whether the current set of goal nodes is acceptable.

Although many different multi-target pathfinding solutions can be applied to the algorithm, we present a particular one to support complexity analysis. This is the same pathfinding strategy used in our implementation. We define a $\text{PositionSearchQueue}(f, \mathbf{c}, E')$ as a queue of tuples $(\mathbf{c} \rightarrow \mathbf{c}', \mathbf{c}', e)$ where $e \in E' \subseteq E$ (we can search for positions targeting a subset of all Enemy Force units if needed). The **new** operation invokes multi-target pathfinding to create a $\text{PositionSearchQueue}$. The goal node test for a search node u succeeds if $E'' \cap f_u(u) \neq \emptyset$, where $E'' \subseteq E'$ is an internal

variable of PositionSearchQueue. Each time the goal node test succeeds, we enqueue new tuples $(\mathbf{c} \rightarrow \mathbf{c}', \mathbf{c}', e) \forall e \in E'' \cap f_U(u)$, where \mathbf{c}' is on u and $\mathbf{c} \rightarrow \mathbf{c}'$ is the route just found. Before resuming pathfinding, we remove $E'' \cap f_U(u)$ from E'' . The search goal test succeeds when $E'' = \emptyset$ or upon the complete exhaustion of search nodes. This approach can be described as “first position found” because the only one tuple is returned for each $e \in E'$, the first one found by multi-point pathfinding. This is a simple and relatively efficient strategy, but it ignores any higher-quality positions just a node or two farther away. The maximum size of a new PositionSearchQueue is

$$N_s(|E'|) = O(|E'|) \quad (32)$$

The complexity of point-to-point and multi-target pathfinding are both $O(v^2)$, where v is the number of search nodes, based on the following argument. We can find a best-scoring path between any pair of nodes using Dijkstra’s algorithm (1959) in $O(v^2)$ time, so clearly this provides an upper bound for point-to-point pathfinding. Now let u be the farthest node at which the goal node test eventually succeeds when multi-target pathfinding is implemented with Dijkstra search. Such a search, by definition, visits all nodes meeting the goal node test and enqueues the appropriate tuples. The extra work to enqueue these $|E|$ tuples is dominated by the $O(v^2)$ pathfinding steps since it is reasonable to assume that the number of search nodes is much greater than the size of the Enemy Force. Define the pathfinding cost for either point-to-point or multi-target pathfinding as

$$T_s(v) = O(v^2) \quad (33)$$

G. COMPLEXITY ANALYSIS

Our goal is to derive a relatively simple expression of the asymptotic running time of the fire support planner using only n , m , and v as measures of the input size. We first combine the running time expressions of the operations and algorithms described above to produce a single, general running time expression using the larger set of variables from

the previous section. We then make some additional assumptions to express these variables as functions of n , m , and v . Finally, we substitute and simplify the resulting expression and convey some insights from the analysis.

1. Expansion of Terms

T_1 is the running time of the fire support planner. We break Equation (17) into an initialization time T_{1I} and a planning loop iteration time T_{1L} :

$$T_1 = T_{1I} + \sum_{i=1}^{mn} T_{1L}$$

where

$$\begin{aligned} T_{1I} &= O(nk_{w1}k_xT_\Psi + k_{A1}T_4(k_{R1}, k_{R1}, k_{w1}, k_{z1})), \\ T_{1L} &= T_2(k_{Pi}) + T_3(k_{Ai}, k_{Pi}, k_{Ri}, k_{wi}, k_{zi}) + O(k_{Ri}) + O(k_{wi}k_{zi}) \end{aligned}$$

It is helpful to have a generic expression for T_4 :

$$\begin{aligned} T_4(k_R, k'_R, k_w, k_z) &= O(k'_R + T_8(v) + N_8(n)k'_RT_5(k_R, k_w, k_z)/n) \\ &= O(k'_R + v^2 + nk'_RT_5(k_R, k_w, k_z)/n) \\ &= O(k'_R + v^2 + k'_R(T_8(v) + nk_xT_\Psi + T_7(k_R, nk_x, nk_x, k_w, k_z))) \\ &= O(k'_R + v^2 + k'_R(v^2 + nk_xT_\Psi + T_7(k_R, nk_x, nk_x, k_w, k_z))), \end{aligned}$$

where

$$\begin{aligned} T_7(k_R, nk_x, nk_x, k_w, k_z) &= O\left(nk_xk_xT_\Psi + k_wT_6(0, 0, k_v, k_z) + nk_x + k_zT_\Psi\left(\frac{k_R - nk_x}{n}\right)\right) \\ &= O\left(nk_x^2T_\Psi + k_w(0 + (0 + k_v)k_zT_\Psi + k_v) + k_zT_\Psi\left(\frac{k_R}{n} - k_x\right)\right) \\ &= O\left(T_\Psi\left(nk_x^2 + k_z\left(k_wk_v + \frac{k_R}{n} - k_x\right)\right)\right); \end{aligned}$$

therefore,

$$\begin{aligned}
& T_4(k_R, k'_R, k_W, k_z) \\
&= O\left(k'_R + v^2 + k'_R \left(v^2 + nk_x T_\Psi + T_\Psi \left(nk_x^2 + k_z \left(k_W k_\nu + \frac{k_R}{n} - k_x \right) \right) \right) \right), \\
& T_4(k_R, k'_R, k_W, k_z) = O\left(k'_R \left(v^2 + T_\Psi \left(nk_x^2 + k_z \left(k_W k_\nu + \frac{k_R}{n} - k_x \right) \right) \right) \right). \tag{34}
\end{aligned}$$

We now expand the initialization expression:

$$\begin{aligned}
T_{1I} &= O(nk_{W1}k_x T_\Psi + k_{A1}T_4(k_{R1}, k_{R1}, k_{W1}, k_{z1})) \\
&= O\left(nk_{W1}k_x T_\Psi + k_{A1}k_{R1} \left(v^2 + T_\Psi \left(nk_x^2 + k_{z1} \left(k_{W1}k_\nu + \frac{k_{R1}}{n} - k_x \right) \right) \right) \right) \tag{35}
\end{aligned}$$

For T_{1L} , we begin by expanding T_3 :

$$\begin{aligned}
& T_3(k_{Ai}, k_{Pi}, k_{Ri}, k_{Wi}, k_{zi}) = \\
& O\left(\frac{k_{Pi}}{n} T_6\left(\frac{k_{Ri} + k_\nu}{n}, k_\Delta, k_\Delta, k_{zi}\right) + k_{Pi} T_6\left(\frac{k_\nu}{n}, k_\Delta, k_\Delta, k_{zi}\right)\right) + \\
& O\left(k_\Delta + 2T_4(k_{Ri}, k_{Ri}, k_{Wi} + 1, k_{zi}) + (k_{Ai} + 1)T_4(k_{Ri}, k_\nu, k_{Wi} + 1, k_{zi})\right) \\
& \text{where } k_\Delta = \frac{k_{Ri}}{n} + k_\nu,
\end{aligned}$$

We expand and combine the T_6 terms:

$$\begin{aligned}
& T_6\left(\frac{k_{Ri} + k_\nu}{n}, k_\Delta, k_\Delta, k_{zi}\right) = O\left(\frac{k_{Ri}}{n} + k_\nu + \left(\frac{k_{Ri} + k_\nu}{n} + k_\nu\right) k_{zi} T_\Psi + \frac{k_{Ri}}{n} + k_\nu\right) \\
&= O\left(\left(\frac{k_{Ri} + k_\nu}{n} + k_\nu\right) k_{zi} T_\Psi\right), \\
& T_6\left(\frac{k_\nu}{n}, k_\Delta, k_\Delta, k_{zi}\right) = O\left(\frac{k_{Ri}}{n} + k_\nu + \left(\frac{k_\nu}{n} + k_\nu\right) k_{zi} T_\Psi + \frac{k_{Ri}}{n} + k_\nu\right) \\
&= O\left(\frac{k_{Ri}}{n} + k_\nu k_{zi} T_\Psi\right),
\end{aligned}$$

$$\begin{aligned}
& O\left(\frac{k_{Pi}}{n}T_6\left(\frac{k_{Ri}+k_\nu}{n},k_\Delta,k_\Delta,k_{zi}\right)+k_{Pi}T_6\left(\frac{k_\nu}{n},k_\Delta,k_\Delta,k_{zi}\right)\right) \\
& = O\left(\frac{k_{Pi}k_{zi}T_\Psi}{n}\left(\frac{k_{Ri}+k_\nu}{n}+k_\nu\right)+k_{Pi}\left(\frac{k_{Ri}}{n}+k_\nu k_{zi}T_\Psi\right)\right) \\
& = O\left(\frac{k_{Pi}k_{Ri}}{n}+k_{Pi}k_{zi}T_\Psi\left(k_\nu+\frac{k_{Ri}}{n^2}\right)\right),
\end{aligned}$$

and then do the same with the T_4 terms, using Equation (34):

$$\begin{aligned}
T_4(k_{Ri},k_{Ri},k_{Wi}+1,k_{zi}) &= O\left(k_{Ri}\left(v^2+T_\Psi\left(nk_x^2+k_{zi}\left(k_{Wi}k_\nu+\frac{k_{Ri}}{n}-k_x\right)\right)\right)\right), \\
T_4(k_{Ri},k_\nu,k_{Wi}+1,k_{zi}) &= O\left(k_\nu\left(v^2+T_\Psi\left(nk_x^2+k_{zi}\left(k_{Wi}k_\nu+\frac{k_{Ri}}{n}-k_x\right)\right)\right)\right), \\
O(k_\Delta+2T_4(k_{Ri},k_{Ri},k_{Wi}+1,k_{zi})+(k_{Ai}+1)T_4(k_{Ri},k_\nu,k_{Wi}+1,k_{zi})) \\
&= \frac{k_{Ri}}{n}+k_\nu+(k_{Ri}+k_{Ai}k_\nu)\left(v^2+T_\Psi\left(nk_x^2+k_{zi}\left(k_{Wi}k_\nu+\frac{k_{Ri}}{n}-k_x\right)\right)\right) \\
&= (k_{Ri}+k_{Ai}k_\nu)\left(v^2+T_\Psi\left(nk_x^2+k_{zi}\left(k_{Wi}k_\nu+\frac{k_{Ri}}{n}-k_x\right)\right)\right)
\end{aligned}$$

Therefore,

$$\begin{aligned}
& T_3(k_{Ai},k_{Pi},k_{Ri},k_{Wi},k_{zi}) \\
& = O\left(\frac{k_{Pi}k_{Ri}}{n}+k_{Pi}k_{zi}T_\Psi\left(k_\nu+\frac{k_{Ri}}{n^2}\right)+(k_{Ri}+k_{Ai}k_\nu)\left(v^2+T_\Psi\left(nk_x^2+k_{zi}\left(k_{Wi}k_\nu+\frac{k_{Ri}}{n}-k_x\right)\right)\right)\right)
\end{aligned}$$

The other terms of T_{1L} are dominated by the T_3 expression, so we have

$$\begin{aligned}
T_{1L} &= T_2(k_{Pi})+T_3(k_{Ai},k_{Pi},k_{Ri},k_{Wi},k_{zi})+O(k_{Ri})+O(k_{Wi}k_{zi}) \\
&= O(k_{Pi}+T_3(k_{Ai},k_{Pi},k_{Ri},k_{Wi},k_{zi})+k_{Ri}+k_{Wi}k_{zi}) \\
T_{1L} &= O(T_3(k_{Ai},k_{Pi},k_{Ri},k_{Wi},k_{zi})) \tag{36}
\end{aligned}$$

2. Additional Assumptions

We assume $k_x = O(\sqrt{v})$: all routes traverse $O(\sqrt{v})$ terrain nodes and have $O(\sqrt{v})$ actions. For a terrain comprised of a uniform $\sqrt{v} \times \sqrt{v}$ grid of nodes, a trip around the entire perimeter of the terrain crosses $4\sqrt{v}$ nodes, so $O(\sqrt{v})$ seems like a reasonable upper bound for the length of a route in a battle plan. This upper bound applies both to the routes of the input plan W_0 and to the route of each fire support task.

We now focus on k_v . Every fire support task w includes a route π and a firing position \mathbf{c} . In the worst case, each action of π is threatened by every $e \in E$. Since $k_x = O(\sqrt{v})$ and $|E| = O(n)$, an upper bound for the number of risk segments needed for π is $O(n\sqrt{v})$. By Equation (6), the maximum number of risk intervals that can be constructed from this many risk segments is $N_B(n, 1, \sqrt{v}) = O(n\sqrt{v})$. For firing position \mathbf{c} , we need only one more risk segment per threat, which does not change the asymptotic number of risk segments or risk intervals for w . The only other source of risk intervals for $w.R_v$ is the route replacement task, if it exists, but this only doubles the upper bound. Therefore, we assume that $k_v = O(n\sqrt{v})$.

We assume that $|W_0| = k_{w1} = O(n)$. In other words, each unit may have only a constant number of routes in the input plan. Normally, we need only one (maximal) route per unit to describe all of its movement in a single battle plan, so this is a reasonable assumption.

Since each unit has a constant number of routes in W_0 , each unit has only a constant number of available time intervals before, after, or between its routes. Assuming that the availability tasks for each unit cover exactly these untasked time intervals, $k_{A1} = O(n)$ as well.

The initial risk set R has up to $N_B(|E|, |W_0|, k_x) = O(n^2 \sqrt{v})$ members, so this is an upper bound for k_{R1} . Each initial member of R has up to $k_x = O(\sqrt{v})$ risk segments, so $k_{z1} = O(\sqrt{v})$.

P is initialized by k_{A1} calls to `GenerateTasks`, each of which generates up to $N_4(k_{R1})$ potential tasks, so $k_{P1} = O(k_{A1} N_8(n) k_{R1} N_5 / n) = O(k_{A1} k_{R1}) = O(n^3 \sqrt{v})$.

Finally, we assume that a single risk segment's risk value can be computed in constant time, so $T_\Psi = O(1)$. As evidence for this, we present an exemplary risk value calculation in Section H. Of course, implementation choices could drive up the asymptotic running time of T_Ψ . This is why we have retained it in the equations up to this point.

3. Results

We substitute these values into T_1 , starting with Equation (35):

$$\begin{aligned} T_{1I} &= O\left(nk_{W1}k_xT_\Psi + k_{A1}k_{R1}\left(v^2 + T_\Psi\left(nk_x^2 + k_{z1}\left(k_{W1}k_v + \frac{k_{R1}}{n} - k_x\right)\right)\right)\right) \\ &= O\left(n^2\sqrt{v} + n^3\sqrt{v}\left(v^2 + nv + \sqrt{v}\left(n^2\sqrt{v} + n\sqrt{v} - \sqrt{v}\right)\right)\right) \\ &= O\left(n^3v^{5/2} + n^5v^{3/2}\right) \end{aligned}$$

To expand the summation $\sum_{i=1}^{mn} T_{1L}$, we need expressions for each indexed variable.

We can derive the following using the $i=1$ values just mentioned and the growth rates in Equations (20)-(23):

$$k_{Ai} = k_{Wi} = O(n+i) \quad (37)$$

$$k_{Ri} = O(n^2\sqrt{v} + in\sqrt{v}) \quad (38)$$

$$k_{Pi} = O(n^3\sqrt{v} + in^2\sqrt{v} + i^2n\sqrt{v}) \quad (39)$$

For k_{zi} , the maximum number of risk segments in any member of R , we already have $k_{z1} = O(\sqrt{v})$. Let w be a new potential task created during iteration i . `StoreAffectedRiskIntervals` is called on w once by `TryGenerateTask`, resulting in $N_7(k_{zi}) = \max(k_x + 2k_{wi}, k_{zi}) + 2 = O(\max(\sqrt{v} + 2(n+i), k_{zi}) + 2)$ risk segments per member of w 's affected risk sets. Also in iteration i , any member of P not selected by `ChooseTask` or removed by `ApplyTask` is modified by `UpdateAffectedRiskIntervals`, resulting in $N_6(k_{zi}) = k_{zi} + 2$ maximum risk segments per affected risk interval. In other words, the members of P have up to two more risk segments than the corresponding members of R . Whenever `ApplyTask` updates R with the affected risk sets of the newly chosen task, which came from P , the maximum number of risk segments in any member of R increases by 2. Therefore,

$$k_{zi} = k_{z,i-1} + 2 = O(\sqrt{v} + i) \quad (40)$$

We expand T_{1L} and terms with respect to i :

$$\begin{aligned}
T_{1L} &= O\left(T_3(k_{Ai}, k_{Pi}, k_{Ri}, k_{Wi}, k_{zi})\right) \\
&= O\left(\frac{k_{Pi}k_{Ri}}{n} + k_{Pi}k_{zi}T_\Psi\left(k_\nu + \frac{k_{Ri}}{n^2}\right) + (k_{Ri} + k_{Ai}k_\nu)\left(v^2 + T_\Psi\left(nk_x^2 + k_{zi}\left(k_{Wi}k_\nu + \frac{k_{Ri}}{n} - k_x\right)\right)\right)\right) \\
&= O\left(\left(n^2\sqrt{v} + in\sqrt{v} + i^2\sqrt{v}\right)\left(n^2\sqrt{v} + in\sqrt{v}\right) + \left(n^3\sqrt{v} + in^2\sqrt{v} + i^2n\sqrt{v}\right)(\sqrt{v} + i)\left(n\sqrt{v} + \sqrt{v} + \frac{i\sqrt{v}}{n}\right)\right) \\
&\quad + O\left(\left(n^2\sqrt{v} + in\sqrt{v} + (n+i)n\sqrt{v}\right)\left(v^2 + nv + (\sqrt{v} + i)\left((n+i)n\sqrt{v} + n\sqrt{v} + i\sqrt{v} - \sqrt{v}\right)\right)\right) \\
&= O\left(\left(n^4v + i^2n^2v + in^3v + i^3nv\right) + \left(n^3v + in^2v + i^2nv + in^3\sqrt{v} + i^2n^2\sqrt{v} + i^3n\sqrt{v}\right)\left(n\sqrt{v} + \sqrt{v} + \frac{i\sqrt{v}}{n}\right)\right) \\
&\quad + O\left(\left(n^2\sqrt{v} + in\sqrt{v} + (n+i)n\sqrt{v}\right)\left(v^2 + nv + \left(n^2v + inv + in^2\sqrt{v} + i^2n\sqrt{v}\right)\right)\right) \\
&= O\left(n^4v^{3/2} + in^3v^{3/2} + i^2n^2v^{3/2} + in^4v + i^2n^3v + i^3n^2v + i^3v^{3/2} + i^4v\right) \\
&\quad + O\left(\left(n^2v^{5/2} + n^4v^{3/2} + in^3v^{3/2} + in^4v + i^2n^3v + inv^{5/2} + i^2n^2v^{3/2} + i^3n^2v\right)\right) \\
&= O\left(n^4v^{3/2} + n^2v^{5/2} + i\left(n^4v + n^3v^{3/2} + nv^{5/2}\right) + i^2\left(n^3v + n^2v^{3/2}\right) + i^3\left(n^2v + v^{3/2}\right) + i^4v\right)
\end{aligned}$$

This allows us to simplify the summation:

$$\begin{aligned}
\sum_{i=1}^{mn} T_{1L} &= O\left(\left(n^4v^{3/2} + n^2v^{5/2}\right)\sum_{i=1}^{mn} 1 + \left(n^4v + n^3v^{3/2} + nv^{5/2}\right)\sum_{i=1}^{mn} i\right) \\
&\quad + O\left(\left(n^3v + n^2v^{3/2}\right)\sum_{i=1}^{mn} i^2 + \left(n^2v + v^{3/2}\right)\sum_{i=1}^{mn} i^3 + v\sum_{i=1}^{mn} i^4\right) \\
\sum_{i=1}^{mn} T_{1L} &= O\left(m^2n^3v^{5/2} + m^3n^5v^{3/2} + m^4n^6v + m^4n^4v^{3/2} + m^5n^5v\right) \tag{41}
\end{aligned}$$

It turns out that all terms of T_{1L} are dominated by a term of $\sum_{i=1}^{mn} T_{1L}$, so Equation (41) is, in fact, an upper bound for the overall complexity of the fire support planner (under the assumptions made in the previous section). A more compactly written, though looser, upper bound is

$$T_1(m, n, v) = O(m^5n^6v^{5/2}) \tag{42}$$

4. Summary of Complexity Analysis

The generation and maintenance of the potential tasks (planning branches) P drives the complexity, which should come as no surprise. The high-order polynomial terms in Equation (42) are perhaps unexpected from a greedy algorithm; they come from the algorithm's attempts to apply every fire support resource against every possible threat and to record all of the effects (affected risk sets) to all parts of the current plan—and then update all of that with every planning branch.

It is important to note that, in most combat models, $v \gg n$ by several orders of magnitude because there are probably many more terrain polygons than there are units. Therefore, even though v has the smallest exponent in the equation, it could still drive the practical running time.

It is beneficial to know that we are in tractable territory with a polynomial running time, but this kind of complexity analysis should not be used as the only “measuring stick” to compare different planning algorithms. Rather, it is meant to help developers and users understand how different, modifiable parameters affect the running time. Some options for reducing the running time of an implementation are

- Reducing m by increasing the minimum allowed suppression duration τ_{\min}
- Reducing n , either by limiting the number of units in the scenario or running multiple fire support planners for subordinate commands
- Reducing v by limiting the size of the terrain, using a coarser terrain representation for the fire support planner (not necessarily for the CSE), or limiting fire support planning to a smaller area of operations (a subset of the terrain)

Since the algorithm works with a best-first scheme, we can simply cut off planning, at the end of a planning loop iteration, after a real world time limit or iteration number. We can be fairly certain that the tasks added were the most important ones, to the extent that Δ is a good predictor of performance.

H. RISK VALUE CALCULATIONS

The definition of a risk value is simply the integral of a piecewise-defined killing rate. This generic description allows us to encompass many potential CSEs, but leaves some work to developers to derive the actual risk value formula. To demonstrate this kind of work, we make some particular assumptions about the combat model and derive a risk value formula in this section.

1. Assumptions

We make the following assumptions about the CSE:

- (1) Units move with piecewise constant velocity, so the equation of motion for any risk segment is of the form $\mathbf{x}(t) = \mathbf{x}_0 + \mathbf{v}(t - t')$ where \mathbf{x}_0, \mathbf{v} are vector constants and t' is the start time of the risk segment
- (2) The killing rate for Enemy Force unit e against Friendly Force unit f is a decreasing linear function of the distance from e to f , or 0 if the distance is greater than maximum range a_{\max}
- (3) No other factors (e.g. derived detection models) affect the killing rate of e against f , including the number of simultaneous targets presented to e
- (4) Suppression of a unit e by a fire support unit f reduces the killing rate of e by a constant coefficient $\beta_{fe} \in (0, 1)$, so the killing rate transformation ϕ of any fire support task is such that $\phi(\psi(t)) = \beta_{fe}\psi(t)$

2. Risk Value for Risk Segments

Given an action (f, \mathbf{x}, t', t'') and threat e at location \mathbf{y} , we construct risk segment $z_{i,j} = (\psi, \mathbf{x}, t', t'')$ where $\psi = K \circ \kappa_{ef}$. Due to assumption (3), we disregard K (it is the identity function). Let $\chi_{ef}(a)$ be e 's killing rate against f as a function of target distance a (rather than time), as described in assumption (2). Since χ_{ef} is a linear decreasing function of range, it must be of the form $\chi_{ef}(a) = \alpha(a_{\max} - a)$ for $a \in [0, a_{\max}]$, where α is a scaling constant. Let $D_{\mathbf{x}} : \mathbb{R}^3 \times \mathbb{R} \rightarrow \mathbb{R}$ be the *distance function for \mathbf{x}* ; given $\mathbf{x}(t) = \mathbf{x}_0 + \mathbf{v}(t - t')$ from assumption (1), let $\mathbf{x}' = \mathbf{x}(t')$ and define

$$D_x(\mathbf{y}, \tau) = \|\mathbf{x}' + \mathbf{v}\tau - \mathbf{y}\| = \sqrt{\|\mathbf{x}' - \mathbf{y}\|^2 + 2(\mathbf{x}' - \mathbf{y}) \cdot \mathbf{v}\tau + \|\mathbf{v}\|^2 \tau^2} \quad (43)$$

for $\tau \in [0, t'' - t']$. This gives us the distance from e (located at \mathbf{y}) to f at τ seconds after the start time t' . We use D_x to translate the killing rate from distance to time:

$$\psi(t) = \kappa_{ef}(t) = \chi_{ef}(D_x(\mathbf{y}, t - t')) = \alpha(a_{\max} - D_x(\mathbf{y}, t - t')) \quad (44)$$

Therefore, the risk value of this risk segment is

$$\begin{aligned} \Psi(z_{i,j}) &= \int_{t'}^{t''} \psi(t) dt = \int_0^{t''-t'} \alpha(a_{\max} - D_x(\mathbf{y}, \tau)) d\tau \\ &= \alpha \left(a_{\max}(t'' - t') - \int_0^{t''-t'} \sqrt{\|\mathbf{x}' - \mathbf{y}\|^2 + 2(\mathbf{x}' - \mathbf{y}) \cdot \mathbf{v}\tau + \|\mathbf{v}\|^2 \tau^2} d\tau \right) \end{aligned} \quad (45)$$

For notational convenience, let

$$\begin{aligned} \tau'' &= t'' - t', \\ \mathbf{x}'' &= \mathbf{x}(t''), \\ a' &= \|\mathbf{x}' - \mathbf{y}\|, \\ a'' &= \|\mathbf{x}'' - \mathbf{y}\|, \\ b &= (\mathbf{x}' - \mathbf{y}) \cdot \mathbf{v}, \\ s &= \|\mathbf{v}\| \end{aligned}$$

We then derive

$$\int_0^{\tau''} D(\tau) d\tau = \frac{1}{2s^3} \left((a'^2 s^2 - b^2) \ln \frac{|s(a'' + s\tau'') + b|}{|a's + b|} + s((s^2 \tau'' + b)a'' - a'b) \right) \quad (46)$$

This is a nontrivial computation, but assuming a finite bit length on the input and output values, there exists a finite upper bound on the computation time. Note that risk segments are connected sequences, so we can reuse the value a'' as a' in the next risk segment, reducing the number of square root computations (since a'' is a distance) by half.

3. Risk Value for Risk Subregions

Since, by assumption (4), the killing rate transformations of fire support tasks are just constant coefficients (i.e. β_{fe}), we can compute the modified risk value of a risk subregion, due to a risk reduction $\lambda(r, w)$, with a single multiplication. If we store the calculated risk value of each risk segment and risk subregion (like the Φ element of a risk interval) and the current risk reduction coefficient of each risk subregion, then we do not actually need to recalculate the risk values of every risk segment. We only need to calculate the risk values of the *subdivided* risk segments.

To update a risk interval rather than create a modified copy, we need only create up to 4 new risk segments (one on either side of the subdivision points) in $4(C + T_\Psi) = O(T_\Psi)$ time, which is actually $O(1)$ based on the previous section. Assume the case requiring two nontrivial subdivisions, resulting in 4 new risk segments. Let $\langle s_x, \dots, s_y \rangle$ be the affected subregion sequence after subdivision has occurred but before risk reduction; s_{x-1}, s_{y+1} are the preceding and following (unaffected) subregions. The subdivided risk subregions, before they were subdivided, were s'_x (i.e. $\langle s'_x \rangle \equiv \langle s_{x-1}, s_x \rangle$) and s'_y (i.e. $\langle s'_y \rangle \equiv \langle s_y, s_{y+1} \rangle$). Let Φ_i be the stored risk value of any risk subregion s_i , and let $\Phi_{i,j}$ be the stored risk value of any risk segment $z_{i,j}$. Φ'_x, ϕ'_x (Φ'_y, ϕ'_y) are the risk value and constant coefficient of s'_x (s'_y), that is, the values before subdivision. Let s_i^* be the risk subregions after their risk has been reduced; Φ_i^*, ϕ_i^* are the corresponding new risk value and coefficient.

The risk value of s_x is $\Phi_x = \phi'_x \sum_{j=1}^{|s_x|} \Phi_{x,j}$. Since subdivision does not change the level of risk (only the representation), the new risk value of s_{x-1} is $\Phi_{x-1} = \Phi'_x - \Phi_x$. Similarly, $\Phi_y = \phi'_y \sum_{j=1}^{|s_y|} \Phi_{y,j}$ and $\Phi_{y+1} = \Phi'_y - \Phi_y$. Given β_{fe} from Assumption (4), each affected risk subregion requires just two multiplications to determine its new data: $\forall x \leq i \leq y : \Phi_i^* = \beta_{fe} \Phi_i, \phi_i^* = \beta_{fe} \phi_i$. The coefficients of s_{x-1}, s_{y+1} do not change. These

calculations require only $|s_x| + |s_y|$ additions or subtractions and $k_s + 2$ multiplications, where $k_s = |\langle s_x, \dots, s_y \rangle|$ is the number of affected risk subregions. Together with the (at most) 4 new risk segment creations, the risk reduction update requires only $O(k_s + k_{xy}T_\Psi)$ where k_{xy} is the number of risk segments in s_x and s_y . This is better than the $O(kT_\Psi)$ time required in general (Equation (11)) since k is the total number of risk segments in the entire risk interval. Even if we need to copy the new risk interval, the running time is $O(k_s + k_{xy}T_\Psi + k)$, which is better than $O(kT_\Psi)$ given the nontrivial computation of Equation (46).

I. EXTENSIONS

We made several simplifying assumptions throughout this chapter. In closing, we discuss some of the implications of relaxing some of these assumptions.

1. Variable Unit Sizes

Using the assumptions of Section H, we can scale the risk reduction coefficient β_{fe} to account for units of different sizes. Let $\beta \in (0,1)$ be the suppression coefficient for a single fire support *entity* in f against a single threat entity in e . We assume that multiple entities combine their risk reduction effects multiplicatively in the same way that multiple units do, so the coefficient for x identical entities firing on one target entity is β^x .

To account for the size of the target, we assume that the *area* covered by the target unit, not its number of entities, is the important variable. For example, two entities sitting shoulder-to-shoulder are just as easy to suppress as a single entity, but a well-dispersed unit requires multiple aimpoints. Let a be the area that a single entity can suppress with coefficient β , let a_e be the area covered by target unit e , and let $y = \frac{a}{a_e}$ be the fractional target area. Increasing the target area decreases y and should result in less effective suppression. Since suppression is *improved* by the power x when there are x

shooters, we make the corresponding assumption that suppression is *degraded* by the power y . Therefore, the coefficient for x entities firing on a target with area y is β^{xy} .

As an example, assume $\beta = 0.5$ and $a = 5\text{m}^2$. If a unit with 3 entities suppresses a unit covering a 20m^2 area, then the effective suppression coefficient is $\beta^{xy} = 0.5^{3 \cdot 5/20} \approx 0.595$.

2. Approximate Risk Values

If one or more killing rate functions cannot be analytically integrated, we could use numerical integration. To do so, we would subdivide each risk segment into smaller risk segments and compute the risk value as if f remained at the segment midpoint (in terms of distance or time) for the risk segment's duration. Using the same assumptions from the previous section, the *approximate risk value* formula is

$$\tilde{\Psi}(z_{i,j}) = \psi\left(\frac{t' + t''}{2}\right)(t'' - t') = \chi_{ef}\left(D_x\left(\mathbf{y}, \frac{t'' - t'}{2}\right)\right)(t'' - t') \quad (47)$$

The maximum allowable length of a segment (that is, the precision of the numerical integral) in this scheme depends on the amount of error we are willing to incur. A brute force approach to finding such a maximum length would be to conduct a binary search on the minimum length constant until we find a pair of values that result in the same planning output; the larger one could then be used. However, this value is not guaranteed to be acceptable on a new map or even a new arrangement of forces on the same map. In some implementations, the maximum segment length resulting from the size of the terrain polygons may already be acceptably small. Using segments much smaller than necessary will result in sufficient precision, but could be wasteful of resources. The tradeoffs between analytical and numerical methods are in many ways analogous to the tradeoffs between discrete event simulations and fixed-time-step simulations (Buss 2011). However, both approaches to risk value calculation can be used in support of either form of simulation.

The following example illustrates how using a numerical integral can complicate the subdivision process. Assume Enemy Force unit e is located at \mathbf{y} . Let z be a risk

segment with midpoint \mathbf{x} , and assume we need to subdivide z at some point not equal to \mathbf{x} . The subdivision results in risk segments z_1 and z_2 , with respective midpoints \mathbf{x}_1 , \mathbf{x}_2 and durations t_1, t_2 . Even with constant linear velocities, we find that, usually, $\tilde{\Psi}(z_1) + \tilde{\Psi}(z_2) \neq \tilde{\Psi}(z)$. As illustrated in Figure 29, the distances $\|\mathbf{x}_1 - \mathbf{y}\|$ and $\|\mathbf{x}_2 - \mathbf{y}\|$ may both be greater than $\|\mathbf{x} - \mathbf{y}\|$, in which case $\mathcal{Y}(z_1)t_1 + \mathcal{Y}(z_2)t_2 < \mathcal{Y}(z)(t_1 + t_2)$. This complicates testing, since we may not be certain that the error is the expected anomaly or some other, more serious problem. In comparison, using analytical integrals always results in $\mathcal{Y}(z_1) + \mathcal{Y}(z_2) = \mathcal{Y}(z)$, subject to floating point precision.

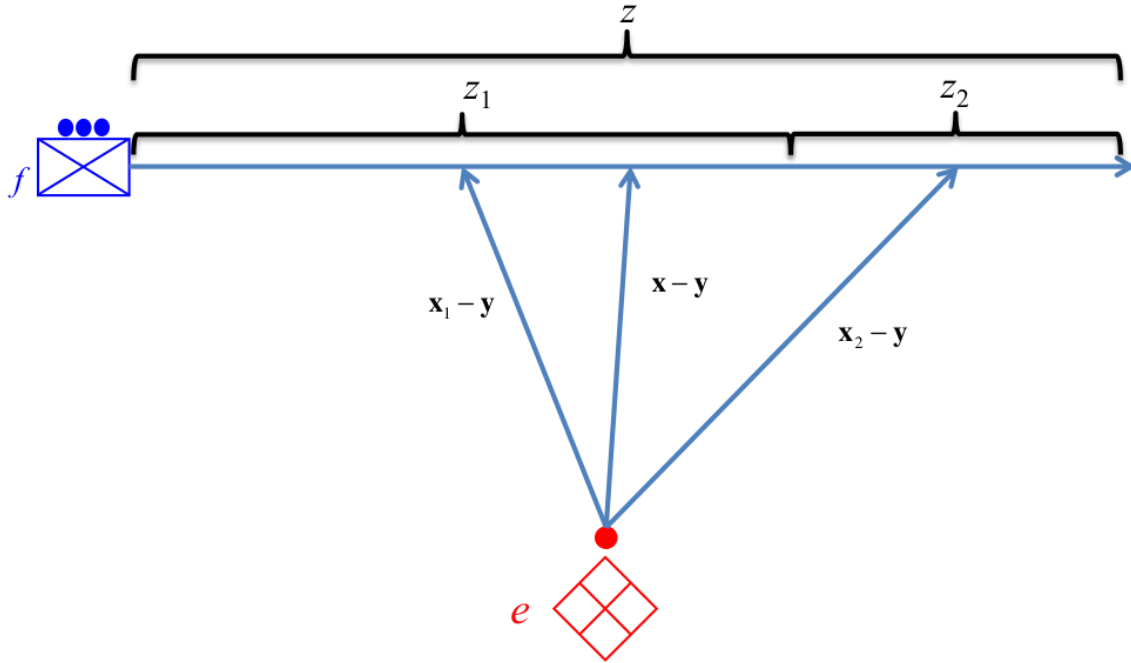


Figure 29. Unequal Risk Segment Subdivision

3. Attrition and Suppression Effects in the Maneuver Plan

We assumed that attrition to the defender is negligible until the assault phase of the attack. If significant attrition to both attacker *and* defender is expected, then we must account for the parabolic effects of attrition as in Lanchester's Square Law (Lanchester 1916). Allowing two-sided attrition in a stochastic model would require revisiting the

integral approach to calculating risk values, since the expected losses to f would be a function not only of target range (as a function of time) but also of the expected size of e (as a function of time). The risk value equations would need to be reworked as a balanced system of differential equations with efficiency functions, not efficiency coefficients (due to the movement of f relative to e) as in the standard two-sided Lanchester model. However, a numerical integral approach in which attrition is only applied at fixed time points may prove sufficient to generate effective plans.

4. Target Selection by Threat Units

If Enemy Force unit e has a choice of several targets—represented by several routes—during some time interval, where will distribute its fire, and therefore where should we assign the risk (expected losses)? The answer depends on the targeting model of the CSE, which may resemble any of the following (not all are mutually exclusive):

- Abstract, deterministic firepower models that distribute fire equally to all possible targets as continuous variables, similar to the efficiency coefficients of the Lanchester equations (1916).
- Stochastic target detection models that test potentially visible targets for detection according to scheduling and ordering rules, and select for engagement the most recently detected target.
- Independent entity-based targeting, such that the members of a unit could end up firing on multiple target units simultaneously.
- Lookup tables parameterized by target distance, target type, and available weapons to choose the “best” target from those currently detected.
- Units that focus the fire of all members on the closest threat unit, or the one with the best chance to destroy.
- Units that choose from a finite set of targeting strategies (scripts) based on repeated forward simulations (of the simulation), such as the Portfolio Greedy Search technique (Churchill and Buro 2013).
- Units that coordinate their fire at a higher echelon according to some policy, such as distributing fire among all targetable units or focusing fire on the perceived greatest threat.

Recall that a risk segment's killing rate is modeled as $\psi = K \circ \kappa_{ef}$. The inner function κ_{ef} is the killing rate of e against f assuming e focuses its entire effort on f . The transformation K , which would actually require more inputs than just the killing rate $\kappa_{ef}(t)$, is a placeholder for the derived detection and target selection models. In Section H, Assumption (3) implies that K is just the identity function ($K = I$). In this simple model, e applies its full firepower to all of its simultaneous targets. This approach is rather simple to describe and implement, but it results in the largest possible set of risk intervals. Assuming that the actual target selection algorithm of the CSE is more complex, $K = I$ overestimates the risk from threats that happen to have a large number of simultaneous targets. However, targeting such threats is a reasonable tactical choice in many circumstances. If developers suspect this approach is leading to unrealistic targeting choices in the fire support plan, they may wish to consider the following three alternatives.

a. *Single Target Approach*

If we know that the threats use a deterministic target selection algorithm, or if we have a closed-form expression for the most likely target at a given time, then we can proceed by predicting the most likely target based on the information at hand. In other words, we define K as a selector function:

$$K_1(\kappa_{ef}(t)) = \begin{cases} \kappa_{ef}(t), & \text{if } e \text{ would target } f \text{ at time } t \\ 0, & \text{otherwise} \end{cases} \quad (48)$$

This results in a subtle but significant complexity: a threat's predicted target could change based only on relative movement of its possible targets, even if all the same targets are still in view. For example, the closest available target could change during the approach to different objectives. A possible danger of using this approach for planning is that the plan we later develop might change our target prediction drastically. For example, by moving some unit f_1 into a firing position, we could inadvertently change the predicted target of, say, e_1 from f_0 to f_1 . That result may be counter to the intended effect

of f_1 's fire or to some other unit's task. This may force us to examine many possible plans in the search for a reasonable one, and it may increase the bookkeeping burden.

b. Probabilistic Multiple Target Approach

The second variant combines $K = I$ and $K = K_1$ by using a probability distribution model for target selection—if one is available, of course. This calculation proceeds similarly to the determination of the most likely target from the single target approach, but we weight the risk value by the probability $P_{ef}(t)$ that e will target f at time t , given all units' expected positions at that time:

$$K_p(\kappa_{ef}(t)) = P_{ef}(t)\kappa_{ef}(t) \quad (49)$$

The $K = K_p$ approach may have similar complexity to $K = K_1$, depending on how P_{ef} is defined. If all probabilities for visible targets are nonzero, then we end up with the same number of risk intervals that we had with $K = I$. However, we should expect to need more risk *segments* because the distribution P_{ef} varies based on the number of targets in view of e , which is not constant over the duration of the plan.

c. Evenly Distributed Multiple Target Approach

The final approach, a special case of $K = K_p$, attempts to strike a balance between complexity and accuracy. Define

$$K_M(\kappa_{ef}(t)) = \frac{\kappa_{ef}(t)}{N_e(t)} \quad (50)$$

where $N_e(t)$ is the number of simultaneous, potential target units available to e at time t . The implication is that all possible targets have an equal chance of being targeted—or that e can equally distribute its firepower over all simultaneous targets. When N_e changes due to a target unit entering or leaving view, we will have to subdivide a risk interval, but we only need visibility calculations to do so—not targeting logic. If we add a route to the plan, we will need to update the values of any preexisting risk intervals that

represent a target simultaneous to the new risk intervals. This is a simple operation; we just multiply each by $\frac{k}{k+1}$, where k is the number of simultaneous targets before the new route was added. Adding a route will also require further subdivisions of the same risk intervals and of the new risk intervals, wherever the number of targets for a threat changes. An appealing feature of this method is that the threats, in the perception of the planner, do not gain firepower simply by having more targets. However, a threat with many possible targets at a particular distance would be weighted the same as a threat with just one target at the same distance, which may not be desirable.

5. Modifying Risk Values

Risk intervals with higher risk values can result in fire support tasks with better scores (because they have more room for reduction), which receive higher planning priority. Therefore, we can affect the behavior of the planner by altering risk values in certain ways. One potential modification is to increase the risk values of risk intervals involving the *main effort*, if the maneuver plan has such a unit designated. The main effort is the commander's bid for success: the unit on which the mission objective most strongly depends, and (by doctrine) the highest priority for support.

To prioritize support for maneuver units over support units, we can use a similar scaling approach. This would allow fire support units to still support each other, but heuristically prefer support of maneuver units. This may not even be necessary with range-based killing rates since maneuver units tend to approach enemy units more closely than fire support units.

Some maneuver objectives are more critical than others. If we have some numerical measure of importance per objective, we can use this to scale up the base risk values of the units assigned to it. The minimum measure—say, for a unit with no maneuver objective, like a fire support unit—should be nonzero. Scaling a base risk value to zero, of course, eliminates it from planning consideration.

Zeroing base risk values can be a useful tool, though. If we want to ignore threats from certain defender units during a time interval, we can subdivide risk intervals at the

endpoints of that interval and reduce the risk values of the contained risk subregions to zero. Since risk intervals, by definition, have nonzero killing rates, this should result in up to two separate (and shorter) risk intervals. By zeroing risk intervals caused by defenders on an objective that is scheduled for an assault (after the assault time), we can model attrition resulting from the maneuver plan. This method has some inherent risk, because the assault might not actually succeed, but it prevents targeting defenders that have already been assaulted.

6. Lasting Suppression Effects and Area Fire

We assumed that each fire support task affects only one unit e , and only during the time $[t_2, t_3]$ that the tasked unit f is firing on e . To model longer-lasting suppression effects (caused, for example, by temporary hearing loss or dust clouds), we could include another time point $t_4 > t_3$ and compute risk reduction over the interval $[t_2, t_4]$. To model area fire—for example, when an artillery sheaf covers multiple unit positions—we could replace the single unit e in fire support tasks with a set $E' \subseteq E$ determined by geometric intersections between unit volumes and the weapon system's expected suppression volume (a derived model). In this case, the killing rate transformation ϕ could be a function of distance from the nearest aimpoints.

7. Interacting Effects

The effects of a combination of fire support tasks may not be equal to the sum (or product) of its parts. There are conflicting arguments on this point. U.S. military combined arms doctrine argues that different types of weapon systems employed simultaneously can have a greater effect on enemy combat power than a homogeneous set of weapon systems separated in time or space. The idea is that the massing of fires with different payloads from many different attack angles creates more fear and confusion and takes away more enemy options during a short-term, violent barrage. If many munitions are impacting around nearby units, they should have a combined psychological effect beyond their individual casualty radii. On the other hand, the killing rate of the suppressed targets cannot be less than 0, so at some point the reduction must

taper off. Planning developers could choose clever functions to model the benefits of interacting effects, but these may not work as expected if the psychology and physics of interacting effects are not represented in the CSE. Development of a fire support planner may call for an upgrade or revision to the CSE's suppression model.

8. Realistic Unit Formations

For combat models with a high level of detail, the calculation of Υ could be complicated by the organization and dispersion of the entities of e and f . For example, when a unit in column formation becomes visible upon emerging from behind a terrain feature, initially only a few unit members are visible. Firepower directed at the unit will initially be focused on the few visible members at the head of the column. Some entities, in different parts of the formation, may never come into view, while some defenders in unfortunate positions may never see any of the attackers traversing a particular route. If the formation is of a significant size with respect to the threats' weapon ranges, then the killing rate against the formation depends on which members are targeted (and how far away they are).

The simplest way to deal with these issues is to ignore them and use the derived model presented at the beginning of this chapter—that is, assuming that all units occupy a single geometric point. If the size of units considered by the planner are very small compared to the length of risk segments, then this assumption may yield reasonable results. If this approach results in too many anomalies, the most exhaustive solution is to treat each entity as a separate unit during risk value computations. This would increase the number of risk intervals in the plan and the number of affected risk intervals per potential task by an order of magnitude, i.e. by a factor equal to the average number of entities in a unit—a significant impact, considering the n^6 term from the complexity analysis.

A compromise solution extends the geometric point approach by describing unit formations as one or two line segments whose vertex locations are determined by the organization and dispersion rules of the planned formations. For example, a *column* formation is defined as one or two entities abreast in the lead rank with the remaining

ranks following in the footsteps of the leaders. This is approximated by a line segment from the lead rank to the trail rank. The *on line* formation is common for the conduct of an assault, and is closely modeled by a line segment perpendicular to the lead (center) entity's direction of movement. The *wedge* formation essentially forms an arrowhead pointing towards the direction of movement, which is close enough to a pair of line segments from the rear-flank edges to the formation leader's location. The purpose of these shapes is to determine not the precise area covered by the formation, but

- The nodes in the annotated mobility graph that it penetrates
- The start and end times of the occupation of each node
- An approximate range of locations, parameterized by time, that threats may target

We could then define a route as a sequence of node sets, marked with each point in time that the node set changes. These changes would occur when of the formation's line segments penetrate a new node and at each point in time that all of its line segments completely exit a node. The implementation would need a more sophisticated definition of the risk value for a risk segment, since a given threat would need to choose a point on the visible portion of the formation to target—for example, the closest point (to maximize the killing rate).

V. IMPLEMENTATION

A. OVERVIEW

We have implemented a prototype ABPS in accordance with the Conceptual Planning Framework presented in Chapter III. This chapter follows the same basic structure, explaining how each part of the framework—Planning Data, Planning Input, and the Plan Generator—is realized in a particular software subsystem. A major focus of the prototype is to demonstrate and test the effectiveness of the Fire Support Planner, whose conceptual model is presented in Chapter IV, so the reader will find the greatest level of detail in the Enhancement Planner subsection of the Plan Generator section. Even more specific notes, such as function inputs and outputs, are included in Appendix B. The remainder of this Overview section discusses the target CSE and high-level software architecture, and it explains some notational conventions for referring to code.

Every ABPS implementation depends on a CSE for its Planning Input and Executable Plan formats. To facilitate prototyping, we have chosen to implement a new, relatively simple CSE in the Unity game development environment (Unity Technologies 2016) rather than work with an existing production or research system. This served as a highly effective risk mitigation measure because it eliminated the need for any prior expertise in a particular CSE. It also ensured that all necessary implementation details were available and well understood during implementation of the planning system.

The Unity interface is well documented and has a robust support community due to its large user base. Its drag-and-drop design, visual debugging capabilities, and the availability of add-on features from the Unity Asset Store resulted in a highly efficient development cycle. The decision to use a custom Unity-based system was informed by discussions with production CSE and Unity users and developers—in fact, the idea originated (in smaller scale) from the prototyping practices of CXXI developer David Reeves, the architect of the CXXI HTN system (see Chapter II, Section E.1).

Our CSE is called Wombat XXI (WXXI). It is fashioned after CXXI in both name and structure. Although it contains minimal features, each component bears intentional

similarity to an element or approach of the CXXI model. A detailed description of the WXXI CSE is presented in Appendix A. This chapter focuses on the ABPS of WXXI.

Behavior development in a game engine prior to implementation in a simulation system is itself the subject of a separate research effort (Miller 2016). The focus of that work is a comparison between behavior trees and CXXI-style HTNs, which on the surface is not relevant since we do not employ either (the HTNs we describe below fit the classical definition more closely than they do the CXXI construct). However, since planning falls under behavioral development—in the sense that planning is a human behavior that we desire to represent in a software simulation—then the planning work done here may be considered a contribution to that broad research area. It requires more time and effort to implement a new planning approach (such as the fire support planner) in CXXI or a similarly complicated system than it does to implement in a simplified system like WXXI. A successful prototype reduces risk for the effort in the production system by identifying challenges at relatively low cost. The more the simplified system resembles the production system, the better the project’s risk reduction. Of course, the resemblance cannot be perfect in the same sense that a model, by definition, is not identical to its simuland.

1. Software Architecture

Figure 30 is a class diagram focused on the ABPS subsystem. It includes some of the key member fields and functions, but the lists are non-exhaustive to reduce clutter. Note that we use the term *member function*, or just *function*, in place of the object-oriented programming term *method* to avoid confusion with the planning methods in the Method Dictionary. Some of the classes referenced in this chapter are defined in Appendix A; we do not attempt to define each lexicographical symbol here, but in many cases, the mnemonic names make definitions obvious to readers familiar with Chapters III and IV.

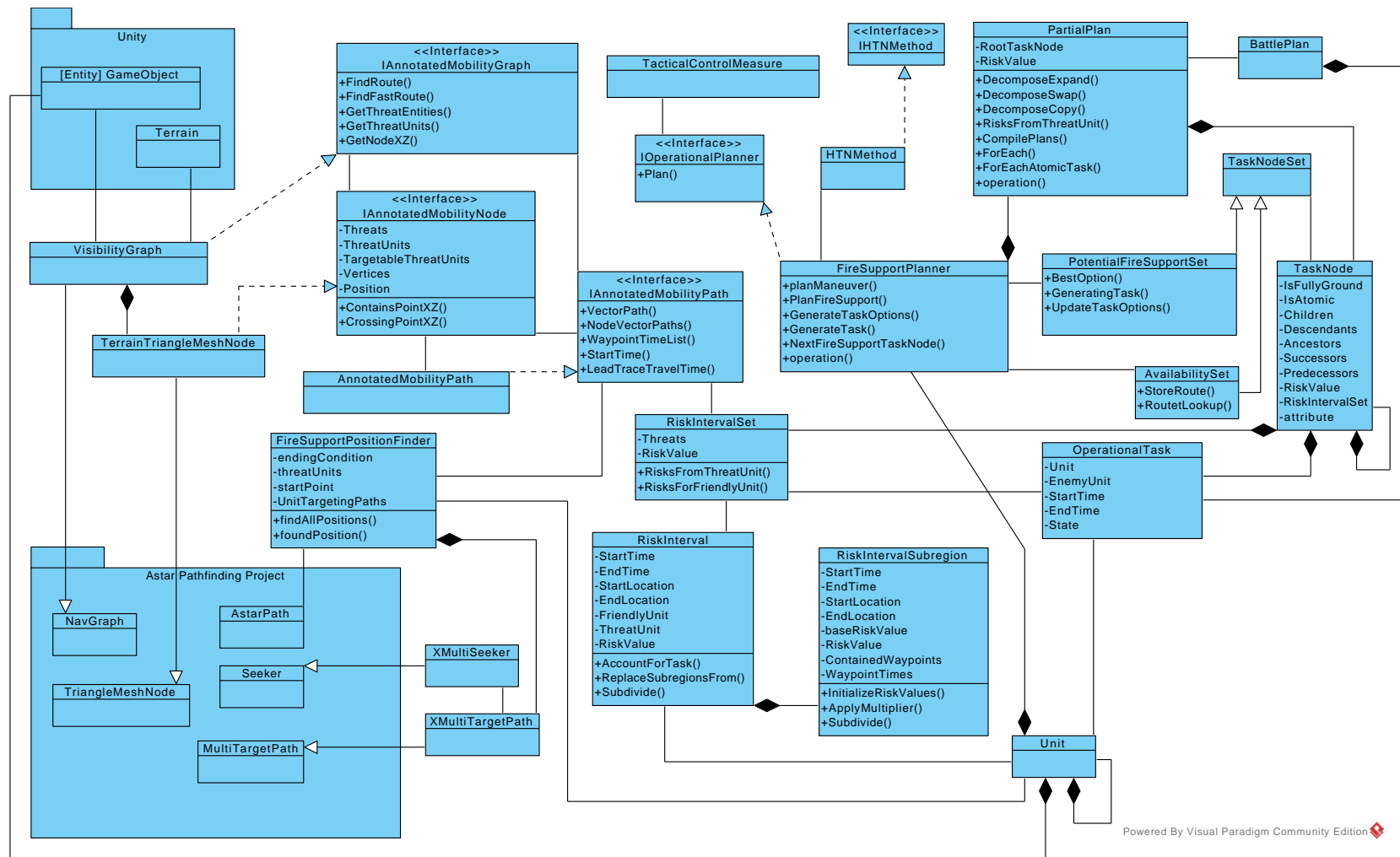


Figure 30. WXXI Automated Battle Planning System Class Diagram

2. Notation

To distinguish C# functions from fields, we use the common parenthetical notation for function signatures. We include the parameter types but not their names, as in

```
TryRouteLookup(EntityInformation, Vector3, Vector3, out IAnnotatedMobilityPath)
```

The function called TryRouteLookup has four parameters: one of type EntityInformation, two of type Vector3, and an output parameter of type IAnnotatedMobilityPath. Where the parameters are not necessary in context, we exclude them and just keep the parentheses, e.g. TryRouteLookup(). Note that C# accessor properties (defined with get {} and set {} keywords) work like parameterless functions but have no parentheses.

When needed for clarity, we prepend the class or struct name to the function name with a period delimiter, such as VisibilityGraph.FindRoute(). In this example, FindRoute() is a member function in the VisibilityGraph class. An actual function call would be written as vg.FindRoute(), where vg is an instance of a VisibilityGraph.

Some of our classes and functions use type parameters. These enclose the symbol for the type in angle brackets. For example, TaskNode<AssaultTask> refers to a specialization of the TaskNode class where the contained OperationalTask is an AssaultTask.

In the OperationalTask class, we sometimes need to leave some task parameters (i.e., member fields) temporarily unspecified. We use standard C# null values for this purpose. For primitive types such as int and float, we use the C# Nullable<T> type to allow a null value, which is otherwise not allowed. The shorthand for Nullable<T> uses the “?” character, for example,

```
float? StartTime
```

is equivalent to

```
Nullable<float> StartTime
```

B. PLANNING DATA

The Planning Data of WXXI's ABPS is mostly focused on the tools needed for fire support planning. The maneuver planning component relies on detailed user input, so it requires little in the way of encoded tactics.

1. Unit Templates

We use the Unity prefab feature as a stand-in for entity and unit templates. A prefab is a Unity GameObject with default attribute settings that is stored in a file separate from the scenario (Unity Scene). Multiple copies of a prefab can be instantiated in a scenario. We use the Unity Transform Hierarchy to subordinate entity GameObjects under unit GameObjects, which allows entire units to be saved as prefabs.

2. Derived Model Dictionary

For expedience of developing the prototype, WXXI has no separate software classes for derived models. Instead, they are coded directly into the fire support planner functions and HTNMethods. Some of these assumptions can be modified in the source code by changing the values of self-documenting constants. A future upgrade could better organize these configurations in a centrally located file or interface, making the Derived Model Dictionary a more coherent component as described in the Conceptual Planning Framework.

Users may create unique unit or entity GameObjects when designing a scenario in Unity's Scene View. The EntityInformation and Weapon classes are designed to support modification of attributes for new unit templates. This provides some of the groundwork for self-configuring derived models (mentioned in Section B.2 of Chapter III). The current version, however, has only been tested for the EntityInformation-BasicInfantry.asset instance, M16A2Weapon class, and M240MachineGun class. The fire support planner does not have sophisticated logic for dealing with units with different weapons. It handles the machinegun squad, which contains two M240MachineGuns and five M16A2Weapons, by considering only the M240MachineGuns.

We present some of the important derived models here to demonstrate how the system reasons about entity-level rules at the unit level. Reactive behavior in WXXI is relatively simple, so not many derived models are needed. If a future upgrade enriches entity-level behavior, a corresponding unit-level modeling update may be needed to maintain the planner’s effectiveness.

a. Derived Movement Models

The ABPS takes advantage of the simple movement model of WXXI, in which entities move at piecewise constant velocity. To predict the location of a unit at some point in the future, the speed of the unit’s initial formation leader is used as the speed of the whole unit. The expected location of the initial formation leader is the assumed location of all unit members at the future point in time for computation purposes. This implementation does not use the “precise” or “compromise” extensions for dealing with realistic unit formations, as discussed in Chapter IV.

Since the movement model of WXXI is based on a fixed time step, the formation leader will overshoot or undershoot its arrival time by up to one time step. By keeping the size of the time step sufficiently small, we avoid the possibility that this will cause a significant difference between the expected and actual arrival time at each waypoint.

(1) Column Formations

For a column formation (the *RangerFileFormation* is the only version in WXXI), using the formation leader’s expected location usually overestimates the risk for an entire unit. In the attack, units tend to be moving *towards* the defenders. The formation leader is the first entity in the formation, and computing risk as if all entities are at the front usually means a shorter distance-per-entity to the threats than if their precise expected locations in the formation were used, such as for threat units A and B in Figure 31. An overestimate is more cautious than an underestimate, so it is the safer option. Computing for a single point is also simpler and less computationally costly than a multiple point approach.

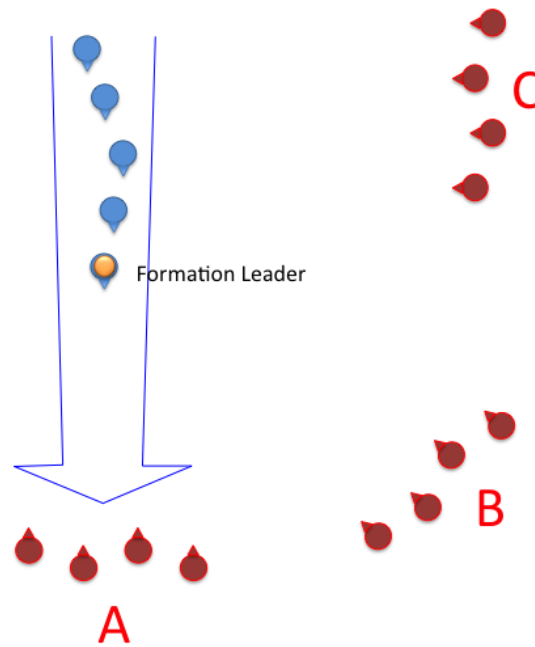


Figure 31. Column Formation With Threats

Once the formation passes a threat unit—for example, unit C of Figure 31—the formation leader is no longer the closest entity. In fact, it is possible for the formation leader to exit an AnnotatedMobilityNode that is threatened by a unit (such as unit C) while most of the formation is still in the threatened node. If this happens, the threat unit will be able to cause casualties during a time interval when there is no corresponding risk interval. This issue is compounded by the decision to use platoon formations for some column movements, which can cause the unit to stretch across several nodes. If the maneuver plan manages to eliminate all intermediary threat units (such as unit C) prior to this situation occurring, then the derived movement model will not degrade the final results. However, a plan that bypasses (a doctrinal tactic used to avoid becoming bogged down by small units) or fails to eliminate some units may suffer from insufficient fire support.

(2) On Line Formations

For an on line formation, all entities are at approximately the same range to threats that are along the direction of movement, such as threat unit A in Figure 32. The formation leader is always at (or moving towards) the lateral midpoint of the formation,

so threats such as unit C that are perpendicular to the direction of movement will have approximately the same number of targets closer to them as they have targets farther away, with respect to the formation leader. Since the width of the formation is usually much less than the range to the threats, units diagonal to the direction of movement (such as unit B) do not stray very far from at least one of these two cases (A or C).

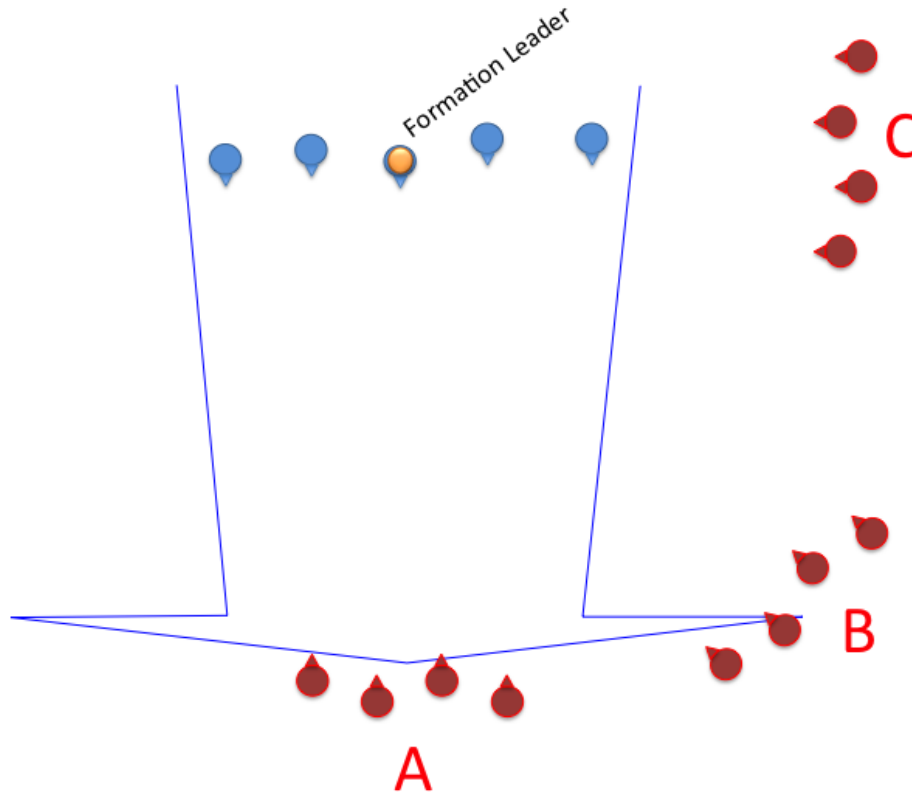


Figure 32. Online Formation with Threats

As with a column formation, it is possible for the formation leader's planned movement to fall within a different node than some of the formation members—this time, the discrepancy is perpendicular to the direction of movement. This is less of an issue for online formations for two reasons. First, by limiting AssaultAction tasks to squads of 13 entities, the formation cannot spread across more than 2, or occasionally 3 nodes. Second, since online formations are only used in the final phase of the attack to cross ground that

has been intentionally selected to offer good fields of fire for the defenders, it is almost never the case that terrain occludes just a fraction of an online formation.

(3) Changing Squad to Online Formation

Although formation adjustments occur at the entity level with respect to Formation object target positions, the maneuver planner always allots 30 seconds of simulation time for squads to change from one type of formation to another. This proved to be sufficient for most situations during functional testing.

(4) Breaking Squads from Platoon Column Formation

For sufficiently large scenarios, the user can employ platoon-level movement in column formation (using the `RangerFileFormation` class). Transitioning squads from a long column into approximately parallel on line formations takes extra time, especially for the rear squad. Platoons have a standard entity count, so their formations have a standard maximum length. The maneuver planner allots 180 additional simulation seconds for platoons to transition from a column to squads on line.

b. Derived Detection Models

Since WXXI has a trivial target detection model, no special data or sophisticated algorithm is needed to determine expected target detection time intervals. Maximum sensor range and line of sight are used to annotate the `VisibilityGraph`. An attacking unit is assumed to be detected by a defending unit immediately and continuously as long as the attacking formation leader's predicted location is within a node visible to any member of the defending unit.

c. Derived Combat Effects Models

The planning system uses four models of the effects of firing (the last of which is vacuous). The mechanics of these reside in the `ReduceRisk()` functions of `OperationalActions` and in the `InitializeRiskValues()` function of `RiskIntervalSubregion`, which are discussed in detail in the Enhancement Planner section below. Only the modeling assumptions are described here.

(1) Attrition Effects of Defender Fires

In accordance with the risk value concept described in Chapter IV, attacking units are assumed to amass casualties as long as their formation leaders reside in nodes that are annotated with one or more defending units. The $K = I$ approach is used, so each defender is assumed to apply its full killing rate to each attacking unit regardless of the number of other units it might be able to target simultaneously. The planner assumes that attacking units are never eliminated, even if their expected losses (RiskValue) exceed their number of starting entities—with the hope that the fire support plan will eventually reduce these losses to the point that the survival assumption holds.

The logic of attrition effects of defender fires is encoded in the RiskValueLinearIntegral() function of TargetHandler (“linear” means the probability of hit is a linear function of distance to the target), and in the RiskIntervalSubregion class. These calculations are parameterized by function calls to the Weapon class, such as getProbKill(), ExpectedRateOfFire, and probHitCoefficient.

(2) Attrition Effects of Assaults

The actual outcome of an assault depends on how many individual entities of the assaulting unit survive the approach to the defending unit’s entity positions and then successfully engage those entities at close range. Both attacker and defender survival depends on stochastic probability-of-hit and probability-of-kill results. The planner uses a much simpler unit-level model: it assumes that each assault will succeed, and that the defending units whose entities are contained within the AssaultObjective’s geometric volume are instantly wiped out at the moment the AssaultAction is scheduled to end. The combination of assaulting unit size, mutually supporting assaults from other tasks in the plan, and suppression effects from the fire support portion of the plan all factor into the actual assault results during replications, so this derived model is not always accurate.

The logic of assault attrition effects is encoded in the ReduceRisk() function of the AssaultAction class (a subclass of OperationalAction). It uses the “zeroing” approach discussed in the Section I.3 of Chapter IV. In other words, if an AssaultAction is planned for time interval $[t_1, t_2]$, AssaultAction.ReduceRisk() subdivides all risk intervals from

the assaulted unit at time t_2 and wipes out all RiskIntervalSubregions occurring thereafter, including those starting after t_2 . As Chapter IV describes, this prevents the planner from wasting effort on “ghost” units that have already been assaulted. The downside of this approach is that there will be no planned suppression after t_2 if the threat manages to survive the planned assault.

During an assault, the entity-level behavior of the attackers (through MoverManager, TargetSensor, and TargetHandler) is to move as close as possible to members of the TargetUnits and fire on them until there are no survivors. Depending on suppression and attrition results, which are stochastic, assaults can take a variable amount of time to conclude with a single surviving side. The assaulting unit often requires additional time, after the formation leader reaches the end of the assault route, to finish off all of the defenders (or be wiped out in the attempt). This is modeled by always adding 180 simulation seconds to the assault route’s arrival time. During functional testing, successful assaults usually finished in less than that amount of time.

(3) Suppression Effects of Defender Fires

Attacking units that receive incoming fire are subject to the same suppression effects as the defenders. However, due to the lethality settings of the weapon models and the fact that mobile attackers are not in protected positions (in contrast to the defenders), effective attrition tends to dominate effective suppression on the attackers. Therefore, the planning system ignores suppression effects on the attacker and focuses instead on attrition (this is the vacuous model mentioned above).

(4) Suppression Effects of Attacker Fires

Only the suppression effects of units acting in a *fire support* role are modeled by the planning system. Suppression effects from *assaulting* units are subsumed by the heavy-handed assault attrition model (see Attrition Effects of Assaults, above). Any suppression from attackers that happen to pass by defenders, or fire reactively while waiting for the next task, are ignored.

Suppression effects due to fire support tasks are applied by calling the `ReduceRisk` function of `SupportByFireAction`, which is designed to fall under a `FireSupportTask` in a hierarchical task network. The details are described in the Enhancement Planner section below, but the basic approach is to assume that each `SupportByFireAction` reduces the killing rate of its target by a fixed, constant fraction from the planned start of firing until the planned end of firing. When more than one unit suppresses a particular target simultaneously, the suppression is assumed to combine multiplicatively during the overlapping time, as described in Section H.3 of Chapter IV.

Due to the assumptions of the suppression model (described in more detail in Appendix A), it takes some time for the members of a unit to enter the `fullySuppressed` state after they begin taking suppressive fire. This depends on many stochastic outcomes in the `TargetHandler` and `Weapon` classes, which in turn depend on the distance from the fire support unit to the targeted unit. However, this time is approximated well enough by a constant value. This value is set in the `maxWarmupTime` field of the `SupportByFireAction` class, causing the fire support planner to start suppression tasks a little earlier than the desired start time, if possible.

d. Special Behavior Models

The ABPS has no other derived models besides those discussed in the three sections above.

3. Task Dictionary

The following tasks, all of which are subclasses of `OperationalTask` or `OperationalAction`, are available to the planning system. The planning system has no way to affect unit behavior except to add tasks to its `PartialPlan` object (which is eventually assembled into a `BattlePlan` for each unit).

Each parameter of an `OperationalTask`—represented by the accessible instance fields of an `OperationalTask` object—allows a null value. This allows the planning system to leave parameters temporarily unspecified during planning. For example, if two tasks are meant to occur in sequence but we do not know the duration of the first task, we

can leave the EndTime of the first task and the StartTime of the second task as null until a later planning step. Some OperationalTasks allow null values for certain parameters in executable tasks, depending on the semantics of the task.

The current set of tasks is described here. For the standard attributes and functions of each OperationalTask, see Appendix A. OperationalTasks are listed first, followed by OperationalActions. Each task name ends with -Task or -Action, respectively.

a. MissionTask

A MissionTask is no more than a generic label for a (relatively) higher-echelon unit. It is used when no more specific OperationalTask exists to describe a desired effect. It can also be used as a parent task for an arbitrary set of tasks for a single unit, such as when several Required Tasks are given to an explicit or implicit top-level unit. In this implementation, we use it for the single company- or battalion-level task and for a squad-level assault (which contains lower-level tasks). A MissionTask has a special field for the mission name that can be set to any string.

b. AvailabilityTask

This is an implementation of the availability tasks described in Chapter IV. AvailabilityTasks have both a precise coordinate location and an IAnnotatedMobilityNode, which must contain the coordinate location. AvailabilityTask has a special member function UnusedPortions(), which takes as input a subinterval of the original task's time interval that will be used for a FireSupportTask. An implementation of the Remainders operation (see Section D.3 of Chapter IV), UnusedPortions() returns the two new AvailabilityTasks than can be made from the portions of the original task's time interval not covered by the arguments. If either unused portion is too short to be useful for a task, that portion is returned as null.

c. AssaultTask

The AssaultTask is used to label a platoon assault that will be decomposed into several more detailed tasks. It has parameters for an AttackPosition, and AssaultPosition, and an AssaultObjective, each of which is a TacticalControlMeasure. It accepts a single

time point for the initiation of the final phase of the assault (called `assaultTime`), which should fall between the task's start and end times.

d. FireSupportTask

`FireSupportTask` is an `OperationalTask`, so it has no effect in the CSE by itself. It must be decomposed into one or more `OperationalActions` (combinations of `FireSupportMoveByRouteAction` and `SupportByFireAction`) to cause units to actually move or fire in WXXI. Due to its importance for the Fire Support Planner implementation, `FireSupportTask` is described in the Enhancement Planner section below.

e. ChangeFormationAction

This `OperationalAction` tells a unit to assume the formation given in the `formationType` parameter, which must be a string matching the name of a subclass of `Formation`. When executed, `ChangeFormationAction` issues an instruction to the entities of the designated unit to remove themselves from any previous `Formation` and arrange themselves in the newly created `Formation`. The details of this behavior are actually encoded in the `Formation` subclass and `MoverManager`. The amount of time to complete the formation change depends on factors such as unit size and prior disposition. The `EndTime` given to `ChangeFormationAction` does not actually determine how long the formation change takes; it works as a planning delay before the next `OperationalAction` can be executed. In other words, it is a derived movement model. If a unit is instructed to move prior to fully establishing its formation, results may not be as expected—for example, an assault might begin with a unit in a partial column formation (an ineffective tactic).

f. MoveByRouteAction

`MoveByRouteAction` instructs a unit to move, in its current formation, along the provided `IAnnotatedMobilityPath`. The `EndTime` is set automatically based on the `LeadTraceTravelTime` of the `IAnnotatedMobilityPath`. The speed and catch-up speed for the movement are set with additional parameters. The speed should match the speed used

to calculate the times of the `IAnnotatedMobilityPath`; otherwise, the `EndTime` will not be set correctly.

g. AssaultAction

`AssaultAction` causes the assigned unit to follow a route, just like `MoveByRouteAction`. Its path, speed, and `catchUpSpeed` parameters work just as they do in `MoveByRoute`. Although very direct routes tend to work best for assaults, `AssaultAction` will accept any `IAnnotatedMobilityPath` given to it. Unlike `MoveByRoute`, `AssaultAction` instructs the unit's Formation to follow the route in `moveAggressively` mode. `AssaultAction` has additional parameters to deal with unique assault behavior:

- `objective`: the `TacticalControlMeasure` that marks the geographical target of the assault. All defending units with at least one entity contained in the objective are added to the assaulting Formation's `TargetUnits` set (which guides the actions of unit members while in `moveAggressively` mode)
- `objectiveNode`: the `IAnnotatedMobilityNode` containing the ending coordinates of the assault path
- `assaultAxis`: the `TacticalControlMeasure` that marks the assault area from start to end, which roughly covers the ground walked by the assaulting formation members. This parameter does not currently influence behavior, but it could be used to designate other `TargetUnits` in a future enhancement.

The entity-level behavior for the assault is controlled by `MoverManager`, `TargetSensor`, and `TargetHandler`. `AssaultAction`'s role is to put those components into the correct mode for the assault. At the scheduled end of the assault, `AssaultAction` puts all Formation members into `moveInFormation` mode (canceling `moveAggressively`) and empties the `TargetUnits` set.

h. FireSupportMoveByRouteAction

This `OperationalAction` is meant to be a child task of `FireSupportTask`. It is identical to `MoveByRoute` (and is a subclass of it), except that its `HasRequiredStartLocation` flag is set to false. This allows the fire support planner to treat `FireSupportMoveByRouteActions` differently than the `MoveByRouteActions` generated by the maneuver planner (which it should not remove or change).

*i. **SupportByFireAction***

This `OperationalAction` is used as the firing part of a `FireSupportTask`. It may be used as a child of a `FireSupportTask`, but it could be used for other purposes as well. It takes as parameters a `SupportByFirePosition` (which is a subclass of `TacticalControlMeasure`) and an `IAnnotatedMobilityNode` that is assumed to contain the `SupportByFirePosition`.

`SupportByFirePosition` has a special `ChangeFormation` member function to switch a given `Formation` into a `SupportByFireFormation`. `SupportByFireAction` uses this function to adjust the fire support unit's entities into their planned positions. It also switches each entity's targeting mode (in the `TargetHandler` module) to suppressing and sets the `TargetUnit` of each entity to match the `EnemyUnit` parameter. This ensures that the fire support unit only engages the target chosen by the fire support planner.

4. Method Dictionary

Although it is not a mandate of the Conceptual Planning Framework or the algorithm presented in Chapter IV, the ABPS uses an HTN data structure for its `PartialPlan`. We could have used a different plan representation, but this choice is best poised to support future enhancements. We anticipate that a more automated maneuver planner would use an HTN (as does the `PlannedAssault` system [van der Sterren 2013]), so working with that data structure shows how a single `PartialPlan` structure can be shared across different Mission Planner and Enhancement Planner components.

The prototype only has a small set of methods. Currently, its maneuver plans are restricted to a few, very specific tactics that depend on some manual user input. This reduces the HTN planning workload on the maneuver planner. The fire support planner's reasoning is encoded in its special algorithm, but it uses the HTN infrastructure to decompose `AvailabilityTask` nodes and `FireSupportTask` nodes.

The HTN infrastructure is described in more detail in the Plan Generator section below, but a basic understanding of a task network is required to understand the method descriptions. The `PartialPlan` contains a tree of `TaskNode` objects, each of which contains a single `OperationalTask` (which may be an `OperationalAction`, since that is a subclass of

OperationalTask). The StartTime and EndTime parameters of each OperationalTask, once bound to constant values, must fall within the StartTime and EndTime parameters of the OperationalTask of the parent TaskNode. Additionally, each TaskNode maintains a set of successors to other TaskNodes. The successor sets of all TaskNodes, when viewed as a relation, form a partial ordering on all OperationalTasks contained in the TaskNodes. When a TaskNode m has a successor n , the semantics require that n 's OperationalTask start no earlier than the EndTime of m 's OperationalTask.

We describe here some of the HTNMethods used by the maneuver planner and fire support planner. Due to the rapid development of the prototype, some of these Methods are “hard-coded” in the maneuver planner, not in subclasses of HTNMethod. Their code could be migrated to a more formal (and reusable) HTNMethod structure with relatively minimal effort.

a. *Create Mission*

To create the initial top-level PartialPlan, the maneuver planner creates a single TaskNode with a MissionTask. It then creates the following TaskNodes, each for a different subordinate unit:

- A MissionTask TaskNode named “Platoon Attack” for each platoon with the isManeuverTaskable flag raised
- A MissionTask TaskNode named “Squad Assault” for each squad with the isManeuverTaskable flag raised
- An AvailabilityTask TaskNode for each unit with the isFireSupportTaskable raised and the isManeuverTaskable flag lowered

It then calls buildManeuverTask() on each of the MissionTask TaskNodes.

b. *buildManeuverTask*

This method-like function expands a MissionTask with a sequence of TaskNodes representing a basic platoon or squad assault. Its details are described in Appendix B. It depends on user position selections, which can give rise to a frontal attack or flanking attack, but there is no special labeling or code for either form of maneuver. The small-

unit assaults may approximate simultaneous or sequential attacks based on the relative directions and distances from assault positions to assault objectives.

c. FireSupportDecompositionMethod

This HTNMethod adds TaskNodes containing OperationalActions as children of a FireSupportTask. These OperationalActions produce the WXXI behavior described by the FireSupportTask. The method generates the following, in the (total) order shown here:

- If the FireSupportTask's InitialLocation differs from its SupportByFireLocation, a ChangeFormationAction to put the tasked command (unit and all subordinate units) into a column formation (RangerFileFormation).
- If the FireSupportTask's InitialLocation differs from its SupportByFireLocation, a FireSupportMoveByRouteAction with the FireSupportTask's RouteToPosition
- A SupportByFireAction targeting EnemyUnit, starting at EffectsTime and ending at EndTime
- If the FireSupportTask's RouteToNextPosition is not null, a ChangeFormationAction to put the FireSupportTask's command into a column formation (RangerFileFormation).
- If the FireSupportTask's RouteToNextPosition is not null, a FireSupportMoveByRouteAction to that position

FireSupportDecompositionMethod decomposes FireSupportTasks *before* they are ready to be added to the PartialPlan. The last two OperationalActions, relating to the RouteToNextPosition, are not really “part of” the FireSupportTask—they are provisional OperationalActions for the immediately following task, should this FireSupportTask be added to the plan. RouteReplacerHTNMethod deals with these provisional tasks (see below).

d. AvailabilityTaskExpanderHTNMethod

AvailabilityTaskExpanderHTNMethod decomposes a TaskNode containing an AvailabilityTask into a totally ordered sequence of TaskNodes. Just as described in Chapter IV (less the TaskNode machinery), this results in an AvailabilityTask, a FireSupportTask, and another AvailabilityTask. The method uses AvailabilityTask's UnusedPortions member function to generate the new, shorter AvailabilityTasks. This

may result in 0, 1, or 2 AvailabilityTasks since short-duration and no-duration tasks are thrown away by UnusedPortions.

To support functional testing, we retain the original AvailabilityTask's TaskNode and add the new AvailabilityTasks and FireSupportTask as child TaskNodes. This allows the entire decomposition tree for each fire support unit to be displayed after planning is complete. The leaf TaskNodes containing AvailabilityTasks describe the availability time intervals for each fire support unit at any given point during the planning process.

*e. **RouteReplacerHTNMethod***

This method is unusual because it performs two different operations on two different TaskNodes. First, it must use its special RemoveRouteReplacementTasks() function to pull the ChangeFormationAction and FireSupportMoveByRouteAction TaskNodes occurring *after* the SupportByFireAction TaskNode—if they exist—out of the source task node. The method temporarily stores these removed TaskNodes in its own container. Later, when the normal Apply() function is used on the destination TaskNode, we remove the ChangeFormationAction and FireSupportMoveByRouteAction TaskNodes occurring *before* its SupportByFireAction TaskNode (which are now obsolete), then replaces them with the stored TaskNodes.

Normally, HTNMethods should not return any of the same TaskNodes (or objects they contain) they were given as arguments, even when some TaskNodes (or contained objects) are left unchanged. This is because a typical HTN plan-space search maintains different versions of the plan. Passing by reference could cause operations on one version of the plan to change parts of other versions of the plan in other planning branches. However, copying a TaskNode (and all of its components) would result in copying all of its components, including its RiskIntervalSet. This implementation uses a hash map between RiskIntervals in the current PartialPlan and provisionally changed RiskIntervals in potentialTasks, and the hash maps depend on the unique object codes of the PartialPlan's RiskIntervals. Replacing those RiskIntervals with copies would break their mappings to all of the updated versions in potentialTasks. Fortunately, the greedy best-first search strategy allows FireSupportPlanner to maintain only *one* explicit PartialPlan,

so the always-copy paradigm is not necessary. RouteReplacerHTNMethod does not copy any TaskNodes; it only moves them from one parent to another.

5. Standard Tactics Configuration Dictionary

Since the prototype ABPS has only five methods, the overhead of creating a tactics configuration infrastructure is not yet worth the configuration management payoff. However, it is still useful to bin maneuver-related methods and fire support-related methods separately.

The prototype does not have a separate data store for Standard Tactics Configurations, so in effect it can only store *one* such configuration for a given unit template: the planning flag settings in that template. Upgrading the maneuver planning component to a more automated capability would likely benefit from an explicit Standard Tactics Configuration Dictionary.

C. PLANNING INPUT

This section describes the Planning Input of the implementation.

1. Map

The Map for a WXXI planning problem is a Unity Terrain object. It is possible to use real world elevation data as the source, and one particular import tool is described in Appendix A. The current version of WXXI does not use any obstacles besides the terrain skin. The potential impact of adding obstacles to enrich the terrain is discussed in the Annotated Mobility Graph section below.

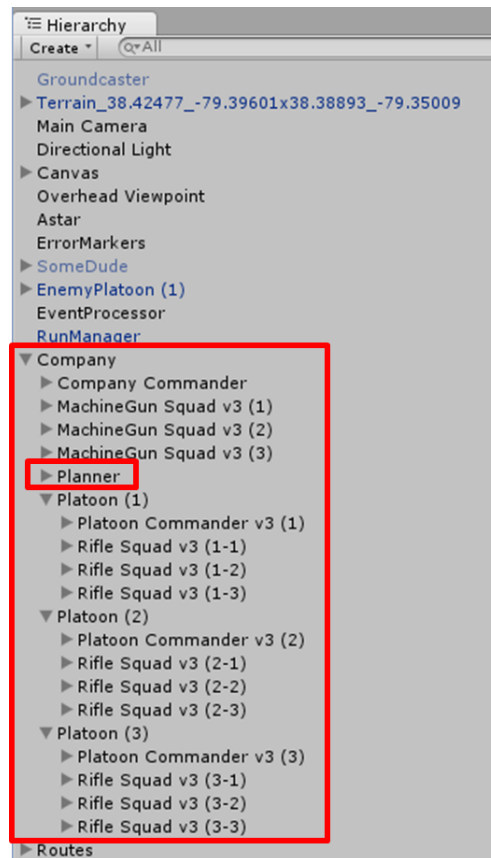
There are two interfaces for Map input. First, the Terrain object must be added to the Unity Scene for a particular scenario. Second, the Terrain object must be provided to the VisibilityGraphEditor, which is an extension of the GraphEditor class from the Astar Pathfinding Project (Granberg 2016). The VisibilityGraphEditor requires some other key inputs, whose purposes are explained in Appendix B.

2. Friendly Force

The commanding unit of the Friendly Force is marked by placing a Planner object under it in the Transform Hierarchy (Figure 33). Every descendant unit of the commanding unit (in the Transform Hierarchy) is treated as a member of the Friendly Force.

a. Unit Organization

Appendix A describes how WXXI takes advantage of the Unity Transform Hierarchy (Figure 33) to organize Units during scenario design. The automated battle planning system only creates tasks for units that have a planning flag raised.



The larger red box shows the Friendly Force. The inner red box shows the location of the Planner GameObject in the Unity Transform Hierarchy.

Figure 33. Friendly Force and Planner in the Unity Hierarchy Window

The maneuver planner operates on platoons and squads. Other echelon arrangements are allowed in the Friendly Force, but are ignored by the maneuver planner. The whole Friendly Force is treated as a collection of platoons and independent squads, with the platoons being comprised of non-independent squads. The user can create the appearance of higher-echelon maneuvers by choosing waypointRoutes that organize movement by company or battalion, but the planner has no symbolic awareness of this level of reasoning.

The fire support planner operates on rifle squads and machinegun squads. In fact, it will generate FireSupportTasks for any unit that has an AvailabilityTask assigned, but it has not been configured or tested for any other unit template.

Squads may be further subdivided into fire teams (for rifle squads) or machinegun teams (for machinegun squads). The maneuver and fire support planner treat squads as atomic units, regardless of the fire team organization, by putting all subordinate entities into a single formation. Organizing the squad formations by fire team could be done with an extension to the Formation class (which is considered part of the CSE, not part of the planning system), with little to no impact on the Plan Generator.

b. Unit Starting Locations

Unit starting locations are set by placing the unit's entities on the Map, that is, on the Terrain in the Unity Scene View interface. Unity offers a rich interface and 3D view to support this. Each Unit object has an attribute called initialFormationLeader; all movement for the unit is based on the location of this entity.

3. Enemy Force

The Enemy Force must be input in two different places: the enemyUnit field of the Planner GameObject's FireSupportPlanner component and the threatUnit field of the VisibilityGraphEditor. In both cases, the entry provides the commanding unit of the Enemy Force. This version of the planning system does not deal with separate enemy commands; it only reasons about the atomic units of the Enemy Force. For our examples, this means four-entity fire teams, two-entity observation posts, and one-entity command

and control units. An Enemy Force is organized in the same way as a Friendly Force, using the Unity Transform hierarchy.

a. Multiple Values

WXXI does not have a Military Intelligence module. The Planner operates with ground truth data on the Enemy Force, so it does not have any alternate Enemy Force values to consider.

b. Plan

The planning system does not have an input for an Enemy Force Plan. It assumes the Enemy Force will defend to the last in its starting positions.

c. Multiple Scenarios

The planning system does not deal with multiple Enemy Force courses of action. Different Enemy Force defensive plans (position selections) can be analyzed separately by copying an entire scenario (Unity Scene) and then modifying Enemy Force positions in the new copy.

4. Other Actors

This implementation does not have any special support for Other Actors. Any number of additional GameObjects with unit or entity components could be added to a scenario separately from the Friendly Force or Enemy Force, but the current planning system would have no way to reason about them (other than line-of-sight blockages).

Some components offer potential insertion points for Other Actors information. In particular, the Annotated Mobility Graph could be augmented by varying the risk or transit time of nodes affected by Other Actors.

5. Tasking

The Tasking information for the ABPS is taken from attributes of Unit and VisibilityGraph objects in the scenario.

a. Required Tasks

The Friendly Force's Required Tasks are provided through the `waypointRoute` attributes of platoons and squads, which allow the key decisions of the maneuver plan to be input by the user. A more automated maneuver planner would accept a set of Tactical Control Measures, such as `AssaultObjectives`, at the Friendly Force level, allowing the Planner to task individual subordinate units for their seizure.

b. Area of Operations

The Friendly Force Area of Operations is assumed to be the entire Map.

c. User Cost Function

The only user-provided User Cost Function information is the `Penalty Weight` attribute of the `VisibilityGraph`. This tells the Pathfinding module the importance of risk versus speed in pathfinding searches (see Pathfinding). The objective function for the fire support planner is currently not user-configurable.

6. Tactics

The prototype leverages the “prefab” infrastructure of Unity for its Unit Templates. The system does not have an explicit data type for planning styles; instead, the interface offers two binary switches for each unit:

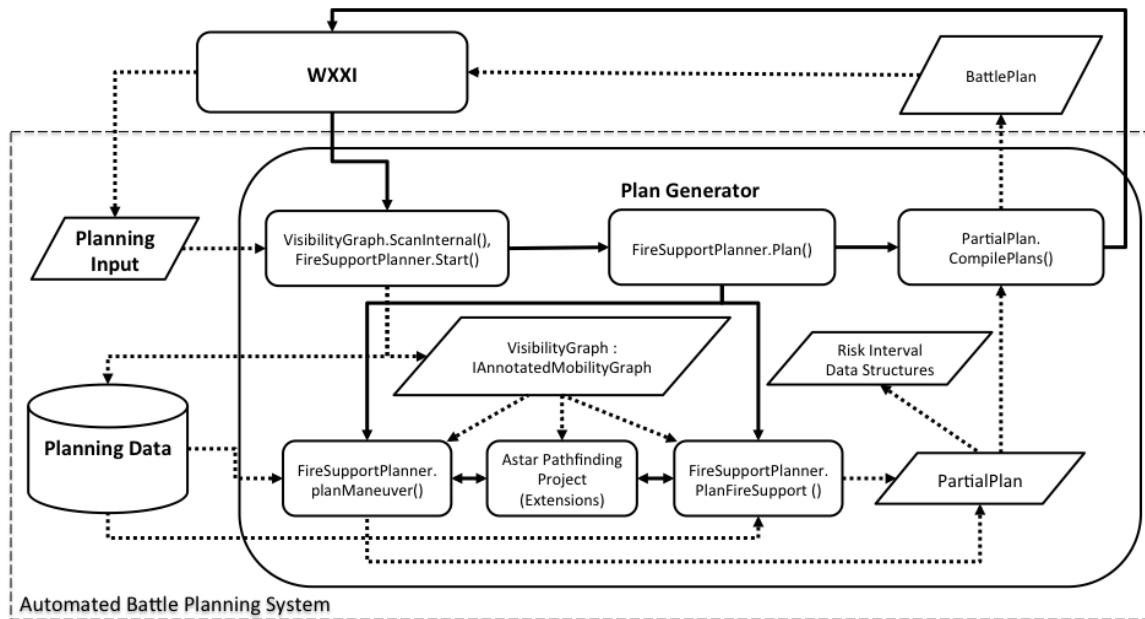
- `isManeuverTaskable`: the planner will use the maneuver-related methods to generate tasks for this unit to assault an objective
- `isFireSupportTaskable`: the planner will use the fire support planner to generate `FireSupportTasks` for this unit after any maneuver-related tasks have finished

We can think of the settings of these two switches for all units in a scenario as a Tactics Configuration. Managing just these two switches across multiple units in multiple scenarios is nontrivial. Explicit Planning Styles and a Standard Tactics Configuration Dictionary would be useful upgrades, especially if the planning capabilities are to be expanded.

D. PLAN GENERATOR

The Plan Generator Implementation (Figure 34) is presented here as outlined in the Conceptual Planning Framework (Chapter III), with one additional component: the Risk Interval Data Structures, which actually cover three software classes.

The reader will find that, for this prototype, not every subcomponent corresponds to a separate software class. For example, the maneuver planner (Mission Planner component) and fire support planner (Enhancement Planner component) are each a member function of a single class called FireSupportPlanner, and the Astar Pathfinding Project (Granberg 2016) is an entire package of classes that support multiple subcomponents,¹⁹ including the Annotated Mobility Graph and Pathfinding module. This specific architecture is motivated, in part, by development speed. Other Framework implementations may choose a different architecture.



Solid lines represent control flow, and dotted lines represent data flow.

Figure 34. Plan Generator Implementation

¹⁹ With extensions written specifically for this planner implementation.

Several of the conceptual components are implemented inside the FireSupportPlanner class. FireSupportPlanner conforms to the IOperationalPlanner API, which includes the Plan() function signature. A more robust Plan Generator could use an interface like this to specify the data exchange formats for its Mission Planners and Enhancement Planners. The prototype is simple enough to be contained in a single class, but refactoring into separate classes would be well worth the effort for any further development.

1. Input Processor

Two separate elements handle input processing. First, as the Unity Scene is loaded for execution, the Astar Pathfinding Project (Granberg 2016) invokes all navigation graph scanners. Our VisibilityGraph is an implementation of that package's NavGraph class, so its ScanInternal() member function is automatically called during this step. Its details are described in the Annotated Mobility Graph section.

Next, FireSupportPlanner's Awake and Start functions are called. These perform the following preliminary steps:

- Determine the commanding unit
- Load relevant HTN methods into an efficient run-time data container. Since there are so few methods, this implementation simply loads all of them, as opposed to filtering a subset into an explicit Tactics Configuration.
- Set up planner-specific data collection objects
- Find the IAnnotatedMobilityGraph that will be used for planning
- Get references to plan visualization tools

Additional input processing is performed in FireSupportPlanner's Plan() function, but control is first relinquished to other WXXI objects to ensure the environment is ready for calls from the Planner. The additional steps are described in the Planning Controller section.

2. Planning Controller

The Planning Controller is invoked when the Unit to which it is assigned calls its `Unit.Plan()` member function. This, in turn, calls the `IOperationalPlanner.Plan()` member function of the Unit's planner field. The function calls are set up in this way to support multiple planning events during a simulation. However, in the current implementation, the Unit containing the `IOperationalPlanner` only schedules the `Plan()` call once, at simulation time 0.0. Invoking the planner from the discrete event system—rather than during a `Start()` function—ensures that all `Awake()` and `Start()` calls for all Unity `MonoBehavior`-derived objects have completed before planning begins.

Before initiating planning, the `Plan()` function finds all subordinate unit objects in the current command hierarchy and stores them in the `subUnits` field. This is done so that, if called during execution, any unavailable subordinate units (due to casualties, reassignment, etc.) are not considered for tasking. This feature makes no difference in the current implementation, since `Plan()` is only called once at the beginning of the simulation, but it would be useful for replanning.

The planning control flow is quite simple: one call to `planManeuver()` followed by one call to `PlanFireSupport(float targetScore)`. For the latter, a fixed `targetScore` of 0.0 is always used, forcing the fire support planner to continue until it runs out of options.

Once both planning functions are complete, `Plan()` displays the `PartialPlan` HTN in Unity's Hierarchy window for debugging purposes. It then invokes the Plan Compiler by calling `PartialPlan.CompilePlans()`, assigning the resulting `BattlePlans` to their corresponding Units, and scheduling `BattlePlan.ExecuteTasks()` with the discrete event system. Finally, it sets up planner-related data collection for the end of the simulation. `Plan()` lets control pass back to the CSE by simply falling through to the calling context.

3. Annotated Mobility Graph

The prototype planning system includes an Annotated Mobility Graph software interface called `IAnnotatedMobilityGraph`, which in turn relies on `IAnnotatedMobilityPath` and `IAnnotatedMobilityNode`. The planner is restricted to these interfaces so that the Annotated Mobility Graph implementation can be entirely replaced,

if needed, with no impact on the source code of other components. This is a particular concern for this subcomponent due to its reliance on Astar Pathfinding Project (Granberg 2016), a software package procured from the Unity Asset Store.

The core of the Annotated Mobility Graph implementation is the NavGraph class of the Astar Pathfinding Project (Granberg 2016). This product is shipped with source code, making it a convenient starting point for the extensions our implementation requires. VisibilityGraph is a custom subclass of NavGraph. It implements IAnnotatedMobilityGraph and some other interfaces required by Astar Pathfinding Project utility functions.

Some of the input for the Annotated Mobility Graph comes from the VisibilityGraphEditor, whose fields are defined in Appendix B. We describe the use of some of these fields throughout the text of this section.

a. Annotated Mobility Nodes

VisibilityGraph uses the triangles of the Unity Terrain as the foundation of its nodes. This concept of triangular nodes follows that of the Astar Pathfinding Project's RecastGraph,²⁰ whose nodes are instances of the TriangleMeshNode class. VisibilityGraph reuses some source code from RecastGraph (which is provided with the Asset Store purchase) but does not derive from that class.

A TriangleMeshNode describes a triangle in three-dimensional fixed-point decimal space²¹ (rather than floating-point space, for performance reasons). The remainder of this section, however, will treat vertices as real numbers for simplicity of description. TriangleMeshNode includes fields and functions that are useful for pathfinding tasks, such as neighbor sets and edge costs. The VisibilityGraph's nodes are instances of our TerrainTriangleMeshNode class, which extends TriangleMeshNode and implements IAnnotatedMobilityNode. IAnnotatedMobilityNode has function signatures for unit and entity visibility information.

²⁰ http://arongranberg.com/astar/docs/graph_types.php#recast

²¹ The Astar Pathfinding Project actually stores vertices as triples of integers, with each unit length in Unity's floating-point space being covered by 1000 integers.

Rather than a flat triangle, we interpret a node as a volume (Figure 35). Let T be a triangle of the Unity Terrain, and let \mathbf{v}_1 , \mathbf{v}_2 , and \mathbf{v}_3 be the vertices of T . Define $\mathbf{u} = (0, H, 0)$ as the “up” vector for entity height H . The Annotated Mobility Node v corresponding to T has the volume V bounded by vertex set $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}'_1, \mathbf{v}'_2, \mathbf{v}'_3\}$, where each $\mathbf{v}'_i = \mathbf{v}_i + \mathbf{u}$. The triangle with vertices $\{\mathbf{v}'_1, \mathbf{v}'_2, \mathbf{v}'_3\}$ is called the *upper triangle* of v , and the triangle with vertices $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$ is called the *lower triangle* of v . If all of a unit f 's entities are contained in V , then a threat entity e whose visibility volumes do not intersect with V cannot target any member of f . This situation defines the statement v is *safe from* e . If “ v is safe from e ” does not hold, then we say e *threatens* v . Even if e threatens v , it may be possible, with more specific information, to find a smaller volume within V that is safe from e —but we do not attempt to find such sub-volumes in this implementation (the Max Subdivisions field of VisibilityGraphEditor is meant to limit the sub-volume search, but is currently not used). This is a conservative approach from the perspective of f in the sense that nodes provide enough information to *avoid* all volumes (of any size) *threatened* by e .²² However, f may not have enough information to *find* all volumes (of any size) *safe* from e . In other words, safe paths through “unsafe” nodes may exist, but the representation does not provide a way to mark them. The total safe volume increases as the terrain resolution becomes finer, but this comes at a performance cost proportional to the number of additional nodes.

²² In fact, this implementation can only claim an approximation of safety, not a guarantee. See the raycasting approach described below.



The upper triangle edges and vertical edges of annotated mobility nodes are shown in bright red. The terrain triangles (lower triangles of annotated mobility nodes) are shown in dark red. An entity (blue) is shown for reference.

Figure 35. Annotated Mobility Node

All entities have the same height in the current version of WXXI, so a constant value can be used for H . The Visibility Height field is set by the user in the VisibilityGraphEditor panel; we keep it set at 2.0 (the height of all entities) for all testing. If entity templates with different heights are needed, then the maximum entity height can be used, with no modifications to the system, for a conservative planning approach. Using multiple heights would require multiple graphs or a more sophisticated node structure.

b. Generating the Graph

Generation of a VisibilityGraph consists of two steps: scanning and annotating. Both are performed by the VisibilityGraph.ScanInternal() function.

During scanning, we iterate through the two-dimensional array of elevation postings in the Terrain object and create two TerrainTriangleMeshNodes per element (except the eastern- and northern-most postings). Nodes sharing a terrain triangle edge are marked as having a connection, meaning that entities are allowed to traverse across that edge. All nodes are inserted into an Astar Pathfinding Project Bounding Box Tree (BBTree) for fast lookup (Granberg 2016). Every vertex is shared by up to six nodes, so the vertices are stored as indices into a shared array of location vectors (objects of type Int3, a utility class of the Astar Pathfinding Project). This storage pattern is similar to an array of triangle vertices for a 3D model (for example, Unity Mesh objects store their vertices as arrays of Vector3 objects, which are floating point location coordinates).

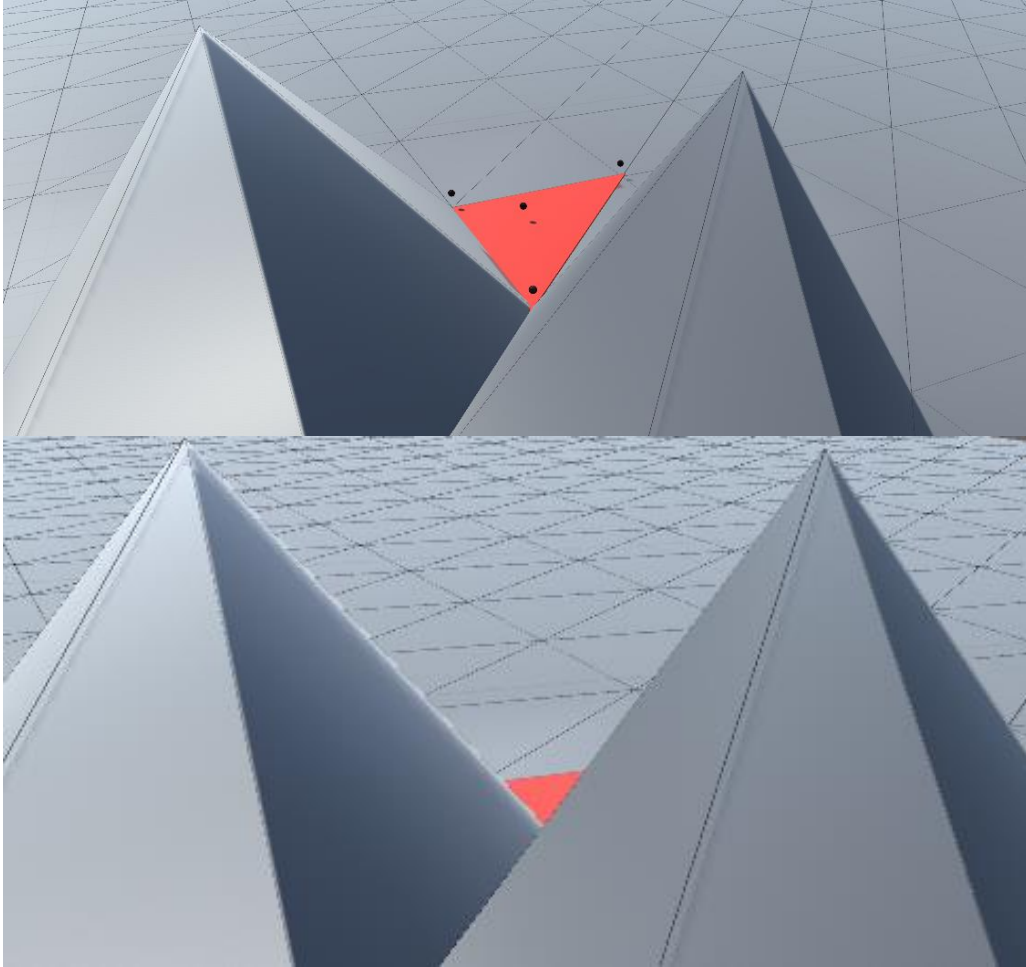
Our graph annotation code checks the visibility of each entity in each unit of the Enemy Force against every upper-triangle vertex (only once per shared vertex). If a raycast from the entity's TargetSensor location to the vertex hits no intervening objects, then all nodes that include the vertex (there can be up to six such nodes) are marked as visible by the entity and by the Unit containing the entity. Note that node volumes are not obstructions themselves; only the terrain triangle collocated with each node's lower triangle blocks line of sight. An additional raycast is made to the geometric *centroid* of each node's upper triangle. The upper triangle centroid is defined as the point whose coordinates are the average of the upper triangle vertices' corresponding coordinates. Similarly, if this raycast hits no intervening object, that single node is marked as visible by the entity and by its Unit. No hierarchical command information is stored in the node's ThreatUnits field, only references to atomic Unit objects (usually fire teams).

A node is marked as visible by an Enemy Force unit even if only one of its entities can trace a ray to just one of the node's vertices. This works reasonably well for the conservative avoidance of threats, but not as reliably for choosing fire support positions. To successfully suppress an Enemy Force unit, the fire support unit should be able to target all (or at least most) of the target unit's members. Further complicating the matter, we cannot predict exactly where the fire support unit's entities will be located. In particular, the machine gunners could end up in different formation positions if their squad takes some casualties prior to the fire support task. To deal with this, we use a slightly different technique to annotate the TargetableThreatUnits for a node. The approach is a rough inverse of the ThreatUnits technique: if an Enemy Force entity is visible from *all* upper triangle vertices and the centroid of a node, then it is considered targetable from that node. If *all* of an Enemy Force unit's members are targetable from a node, then the unit is considered targetable from that node. Note that the Enemy Force unit need not be contained in a single node to be considered targetable—the reasoning is from the targeting node, not from the threat-containing nodes. On some Terrains, it is possible to place a unit's entities such that no node (except some that are dangerously close, perhaps) meets these conditions. This would result in an Enemy Force unit that

cannot be targeted by the fire support planner (at reasonable cost), but it turns out to be very difficult to contrive such a placement of Enemy Force entities.

The “upper triangle” approach just described takes advantage of the simplicity of WXXI terrain. Since the terrain skin has no holes or overhangs (such as windows, awnings, or arched rock formations), whenever we can trace a ray from an observer point above the terrain skin to a point inside a node’s volume, we can also trace a ray to a point on its upper triangle. The following argument supports this claim. If the original penetrating ray does not already pass through the upper triangle, then it must pass under it. We can elevate the aimpoint (keeping the source point fixed) until the ray passes through the upper triangle. The ray, with its original aimpoint, already lies on or above every terrain triangle that it crosses or touches in the xz plane, so raising the aimpoint (which does not change the ray’s projection onto the xz plane) cannot cause it to intersect any new part of the terrain.

It is possible that a node v marked safe from unit e by this implementation is not actually safe. In the lower image of Figure 36, for example, an observer collocated with the camera has visibility into part of the red node’s volume (and part of its upper triangle) but cannot trace an unobstructed ray to any of its upper triangle vertices or centroid. Despite the imperfection of the technique, it works reasonably well for tactical planning and has constant running time per potential threat entity for a fixed Terrain size (one raycast from each entity to each vertex). A different technique, such as taking advantage of GPU power with depth maps (Darken 2004), could be used during annotation with no impact on the source code of other components.



In the first image, we can see all of the node's upper triangle corners and centroid (marked with small dark spheres). In the second, from a lower-altitude viewing position, we can see part of the node but none of the upper triangle corners or centroid.

Figure 36. Two Different Perspective Views of a Node

Once all nodes are annotated, their traversal penalties are set. We iterate through each entity that threatens each node, adding a special risk value to the node's total penalty. Using each threat entity's `TargetHandler` component, we call the `RiskIntervalValue` function to compute expected losses. The targeting range for this computation is taken as the range from the threatening entity to the centroid of the node's lower triangle. The duration is given as the time for the *sample unit*—the small unit referenced in the `VisibilityGraphEditor`'s `Friendly Unit` field—to traverse the longest

edge of the node's terrain triangle at normal traveling speed. This is guaranteed not to underestimate the expected losses caused by the threat entity to a single target traversing the node with constant linear velocity because no linear traversal of the node could take longer than a traversal of its longest edge. Note that node triangle edges have different lengths because they depend on the relative y-coordinates (elevations) of the terrain triangle vertices.

4. Pathfinding

Pathfinding for the planning system uses the Astar Pathfinding Project (Granberg 2016) as its foundation. Many of the features of this package operate on its abstract NavGraph class. VisibilityGraph is derived from NavGraph, so the A* search implementation is able to work with it.

a. Cost Functions

We use tactical pathfinding, meaning that the cost of a step in the graph search is a linear combination of distance and risk (van der Sterren 2002). The Astar Pathfinding Project's path cost (*g-cost* in AI search terminology) is structured to include *connection costs*, associated with a pair of nodes, and *node penalties*, associated with single nodes (Granberg 2016). The cost of a path is the sum of the connection costs and the node penalties for the sequence of nodes it contains. We store the distance part of the cost data as the connection cost (the default representation of the package) and the sum of special risk values (described at the end of the previous section) for all entities threatening a node as that node's penalty.

All node penalties are multiplied by the Penalty Weight field of VisibilityGraphEditor. This allows the user to scale the relative importance of risk versus distance. The same number also serves as a unit conversion factor. Since risk is measured in expected lost entities and distance is measured in millimeters (Int3 coordinates have 1000 integers per floating point unit, and we interpret one unit as one meter), a rather large value is needed for risk to play a significant role in pathfinding. We use a default value of 25,000.

For *fast path* searches, the node penalties are ignored and the distance (connection costs) alone is used. Fast paths are discussed in more detail below.

For point-to-point search, the system uses the Astar Pathfinding Project's built-in Euclidean distance heuristic (Granberg 2016), while multi-point search uses the zero heuristic (i.e., uniform cost search).

b. Invoking a Search

Pathfinding is initiated through the `IAnnotatedMobilityGraph` interface for point-to-point search or through `FireSupportPositionFinder` for multi-point search. `IAnnotatedMobilityGraph` offers two pathfinding functions: `FindRoute()`, which initiates a normal tactical pathfinding search, and `FindFastRoute()`, which initiates a search using the distance-only cost function. A `FireSupportPositionFinder` actually performs its pathfinding immediately upon object construction.

The Astar Pathfinding Project (Granberg 2016) is capable of spreading pathfinding calculation steps across multiple game rendering frames, but the planning system is implemented to work in a single frame. We force pathfinding calls to run in the foreground by calling the `AstarPath.WaitForPath()` function. This call is specific to the Pathfinding implementation, so it is hidden from non-Pathfinding planner components like `FireSupportPlanner`. It can be found in `VisibilityGraph.FindRoute()`, `VisibilityGraph.FindFastRoute()`, and `FireSupportPositionFinder.findAllPositions()`.

The `WaitForPath()` function requires a `Seeker` object to look up modifiers (see below) and other settings, and it places the result of the pathfinding search (a `Path` object) in the `Seeker`. Both `Path` and `Seeker` are defined in Astar Pathfinding Project (Granberg 2016), so we also hide these classes from other components. Since the implementation only invokes one pathfinding search at a time, we could make do with one shared `Seeker`. For extensibility, though, all potential formation-leading entities are given their own `Seeker` objects, and each pathfinding call uses the `Seeker` of the initial formation leader of the relevant unit. In future extensions to the ABPS, these individual `Seekers` could be customized to vary pathfinding results. For example, a unit with vehicles may need to use

a RadiusModifier with different parameter settings to prevent its entities from encroaching into adjacent nodes.

A WaitForPath() call expects a Seeker object, but the Conceptual Planning Framework is defined in terms of units (instances of the Unit class in this implementation). The translation works as follows: FireSupportPlanner calls FindRoute(), FindFastRoute(), or constructs a FireSupportPositionFinder, passing a unit as a parameter. The called function or constructor uses Unit.InitialFormationLeader to get a reference to the formation leader's Seeker. That Seeker is used in the AstarPath.WaitForPath() call, and the result of the search (a Path) is placed in the Seeker. Finally, the Path is used to construct an AnnotatedMobilityPath (described in the next section), which is returned as an IAnnotatedMobilityPath. In the case of FireSupportPositionFinder, rather than a return value, an enumeration of KeyValuePairs is accessible from the UnitTargetingPaths property. Each pair contains a threat unit and an IAnnotatedMobilityPath to a position that can be used to target it.

c. Annotated Mobility Path

The IAnnotatedMobilityPath interface is used as a layer of indirection between the Pathfinding component and other components of the Plan Generator. In this implementation, IAnnotatedMobilityPath is implemented by the AnnotatedMobilityPath class, whose constructor takes a Path (a class defined in Astar Pathfinding Project [Granberg 2016]) object as its primary parameter. Path can provide a list of Vector3 coordinates or a list of GraphNode objects as representations of the route it describes.

In addition to the locations and nodes that it traverses, IAnnotatedMobilityPath stores the start time and waypoint crossing times of its route. AnnotatedMobilityPath gets these values from the startTime and speed parameters of its constructor, assuming constant linear velocity between waypoints.

Although FireSupportPositionFinder offers multi-point pathfinding, an IAnnotatedMobilityPath can only provide a single point-to-point route. This is why FireSupportPositionFinder provides multiple IAnnotatedMobilityPaths via the UnitTargetingPaths property.

d. Point-to-Point Paths

The Astar Pathfinding Project (Granberg 2016) supports point-to-point pathfinding with its built-in classes. ABPath, an extension of Path, is the standard point-to-point path class.

The only difference between tactical paths returned by VisibilityGraph.FindRoute() and “fast” paths returned by VisibilityGraph.FindFastRoute() is that the base risk values stored in the node penalties are ignored in the latter. “Fast” paths are useful for the final phase of an assault, where a meandering path (attempting to avoid visibility to threats as much as possible) would only increase the time spent closing on the objective and suffering close-range fire from the defenders.

e. Multi-point Paths

The multi-point pathfinding required by the fire support planner has two features that distinguish it from point-to-point pathfinding: a nontrivial goal test (rather than searching for a particular node) and a search that continues until several acceptable goals have been found. The first feature is offered by the XPath class of the AstarPathfindingProject, which allows a developer to define an arbitrary goal test function. The second feature is offered by the MultiTargetPath class, which is designed to find paths to several nodes in a single search. Unfortunately, MultiTargetPath does not offer user-defined goal tests like XPath, so we combine these features into a custom class called XMultiTargetPath.

Much of the MultiTargetPath code needed to be re-implemented in XMultiTargetPath due to private function definitions in the former. The source code access provided by the Astar Pathfinding Project (Granberg 2016) was vital for this process. By working with a subclass of MultiTargetPath, FireSupportPositionFinder can use the Seeker.StartMultiTargetPath() function with no changes to the Seeker class. This call occurs during FireSupportPositionFinder construction, in the findAllPositions() function.

FireSupportPositionFinder accepts a set of Unit objects in its threatUnits parameter. It maintains a FireSupportEndingCondition (a subclass of Astar Pathfinding Project's EndingCondition) as the goal test for each node visited in the search. Each time the set intersection of a visited node's TargetableThreatUnits annotation and threatUnits is nonempty, the node is marked as a goal. An AnnotatedMobilityPath is created from the start point to the center of the goal node using the XMultiTargetPath's search path, and a mapping is created from each matching targetable Unit (that is, each member of the intersection) to that AnnotatedMobilityPath. Finally, the matching targetable Units are removed from threatUnits and the search continues. Once threatUnits is empty or all nodes have been visited, the search ends.

FireSupportPositionFinder presents a property called UnitTargetingPaths to allow access to the mappings created during the search. These are returned as an IEnumerable of KeyValuePair objects, each of which contains a Unit and an IAnnotatedMobilityPath.

f. Postprocessing

The point-to-point pathfinding results from the Astar Pathfinding Project (Granberg 2016) are returned as ABPath objects. Two key object properties provide the navigation information for the path: ABPath.Path is the sequence of nodes, and ABPath.waypointPath is the sequence of location coordinates. During the search, the package uses line segments connecting the lower triangle centroids of adjacent nodes for its distance calculations. The initial search result is an ABPath with one ABPath.waypointPath element per ABPath.Path element: the lower triangle centroid of each node. An entity or formation traversing such a default path has an unnatural appearance and covers more ground than it needs to. A zigzag-like path through centroids (Figure 37) takes longer to traverse than a more direct path (not passing through centroids) through the same sequence of nodes (Figure 38).

encroaching on adjacent node volumes that may be less safe than the ones selected for the path—but only if the selected radius contains the entire width of the formation. The basic approach of the Radius Modifier algorithm is to add waypoints to the WaypointPath so that line segments maintain a minimum distance from all non-portal edges. Our entities have an actual radius of 0.5m, but we use a Radius Modifier setting of 4m. This easily contains a column formation, but does not entirely contain a squad on line formation. Since speed is more important than node containment during an assault, we accept some error for the on line formations. Radius Modifier’s other parameter, called Detail, determines the maximum number of additional points that can be added to “smooth out” sharp turns. Since highly detailed aesthetics are not an objective for the prototype, we use a setting of 2 for Detail.

FunnelModifier straightens paths by adding and removing waypoints from the WaypointPath to create the shortest possible path through the same sequence of nodes. This effectively removes the unnatural appearance of zigzagging through centroids. The Astar Pathfinding Project’s Funnel Modifier is an implementation of the Funnel Algorithm (Mononen 2010). It does not take any parameters.

The Modifiers operate on AStar Pathfinding Project’s Path objects prior to creation of our AnnotatedMobilityPath objects. Once Modifiers are applied, the WaypointPath does not have any direct mapping to the node sequence. In some cases, the line segment between a pair of waypoints may cover a large number of nodes due to the straightening capability of the Funnel Modifier. In other cases, the RadiusModifier may have added several waypoints in a single node to keep a sharp turn away from a non-portal edge. Since the RiskInterval class will need the precise time that Units enter different nodes, AnnotatedMobilityPath inserts a new waypoint for each intersection (in the xz plane) of the path with a portal. This is not a Modifier—that is, the code is not in a subclass of the Modifier class and it does not change the ABPath object. The additional points are added during creation of the AnnotatedMobilityPath.

The IAnnotatedMobilityPath interface includes a node-specific waypoint accessor via the NodeVectorPaths property. This is a list of lists, where the outer list corresponds to the node sequence and the inner lists are the waypoint sequences for individual nodes.

There is redundancy in this approach because each final waypoint of an inner list is identical to the first element of the next inner list. However, this simplifies some of the code that operates on NodeVectorPaths. As a result of the design, every inner list has at least two waypoints: the entrance point and exit point of that node.

5. Partial Plans

The partial plan representation of this prototype is a custom-built HTN. The Fire Support Planner algorithm, in the form presented in Chapter IV, does not actually require an HTN or reason at multiple echelons. However, we use multi-echelon maneuver plans for the test scenarios to help demonstrate and test its usefulness in a hierarchical planning context. One of the best examples of a functional multi-echelon maneuver planner (van der Sterren 2013) uses HTNs, so operating within that paradigm helps demonstrate how multiple techniques can be combined as advocated in the Conceptual Planning Framework.

This section explains the concepts and design principles of the software classes that comprise the partial plan. More details, such as specific functions and properties, are provided in Appendix B.

a. First-Class Planning Objects

There is a potential for confusion when introducing the HTN paradigm to the military domain because it results in two competing hierarchies: the command hierarchy (the organization of the Friendly Force) and the task hierarchy (the decomposition tree of a PartialPlan). One helpful observation is that the task hierarchy can enrich, but not violate, the command hierarchy. By enrich, we mean that a task for one unit may be decomposed into several, more detailed tasks for the same unit. This increases the number of layers in the plan hierarchy beyond what we find in the command hierarchy. We define a *command hierarchy violation* as an attempt to decompose a task for a unit into a task for a senior unit in the command hierarchy. Given a hierarchical plan display, command hierarchy violations are easy to catch. In this way, unit semantics help with error checking.

Every `OperationalTask` has a unit as one of its parameters, but the nature of the military domain demands that we treat units as more than just “another parameter.” Military action does not have a plausible explanation unless the initiating orders and sustaining actions flow down through the unit organization. In military HTN planning, we say that tasks (`TaskNodes`), methods (`HTNMethods`), and units (`Units`) are the three types of first-class objects.²⁴

b. Task Nodes

The `TaskNode` class corresponds to the task object of a traditional HTN planning system. `TaskNodes` de-couple the HTN machinery from `OperationalTasks` (and `OperationalActions`), which are meant to describe the activities of individual units. A generic-type subclass of `TaskNode`, `TaskNode<T>` where `T` is an `OperationalTask`, allows developers to specify operations on specific kinds of `TaskNodes`.

Each `TaskNode` contains an `OperationalTask`, and the `OperationalTask` is ignorant of its containing `TaskNode`. This lets us define `OperationalTasks` without making assumptions about the planning approach. For example, we could use the same `OperationalTasks` in a hierarchical GOAP planner (Pittman 2008). This separation also simplifies the Plan Compiler because the contained `OperationalActions` can be removed from the `TaskNodes` and placed directly into `BattlePlan` containers; they are already in the executable format. (In fact, all `OperationalTasks` are placed into `BattlePlans` for runtime logging, but only `OperationalActions` affect the state of the CSE.)

The only type of temporal relationship between `TaskNodes` in the current implementation is predecessor/successor, so our HTNs could be described as STNs (Ghallab, Nau, and Traverso 2004, 231–244). Other kinds of temporal relationships could be added with additional `TaskNode` properties to increase the generality of the infrastructure.

²⁴ We are making this point here, rather than in Chapter III, because the Conceptual Planning Framework does not assume an HTN planning approach.

c. PartialPlan Class

The `PartialPlan` class has a reference to a root `TaskNode`. All of the `PartialPlan`'s details are contained in the `TaskNode` hierarchy at and below the root. `PartialPlan` provides a few additional member properties and functions to facilitate planning operations. These are described in Appendix B.

`PartialPlan` currently supports only the *expand* paradigm for HTN planning. Expanding means retaining the decomposition tree; each HTN method adds subordinate tasks under some existing task in a task network. Temporal relationships between the expanded task and other preexisting tasks must be implicitly or explicitly passed down to the newly added tasks (we use the implicit approach).

In the *replace* paradigm, an alternative to the expand paradigm, HTN methods remove tasks from a task network and replace them with other tasks. This approach would require explicit translation of temporal relationships from the removed task to the new tasks.

d. HTN Methods

An HTN method may be created by implementing the `IHTNMethod` interface. Its two function signatures are `MeetsConstraints()` and `Apply()`. `MeetsConstraints()` returns true if it is given a `TaskNode` that it can decompose, and `Apply()` performs the decomposition.

The `HTNMethod` class allows behavior developers to create methods as data objects rather than subclasses. This approach takes advantage of the C# delegate construct. `HTNMethod` is designed with the assumption that every method has a name, a small amount of unique executable code, and nothing else. Normally, this situation—where only one instance of each “type” is ever needed—would call for static classes, but such an approach would involve a significant number of class declarations once the Method Dictionary grows to a reasonable size.

The `HTNMethod` constructor accepts delegates containing the unique executable code for the method. In the prototype, the `HTNMethods` are constructed in a static code

block and placed in an efficient-lookup data structure; future improvements could read and write HTNMethod instances from persistent storage. Despite the availability of the single-class approach, explicit subclasses of HTNMethod are still allowed. This could be useful for implementing Method Sets with variable binding constraints.

In fact, this prototype has only one direct instance of HTNMethod (FireSupportDecompositionMethod). The Create Mission and buildManeuverTask() function described above in the Method Dictionary section are not HTNMethod instances—they are actually functions of FireSupportPlanner that work much like HTNMethods. This design is only due to the heavily user-defined maneuver plan input. A system offering more automated maneuver planning would benefit from translating Create Mission, buildManeuverTask(), and other methods to HTNMethod objects and placing them together in a single container.

The Fire Support Planner algorithm, due primarily to its greedy design, lends itself to some expedient techniques that are not supported by traditional HTN planning—however, the use of so-called *critics* to make arbitrary changes to an HTN is well-established (Ghallab, Nau, and Traverso 2004, 249–250). Some of these are implemented through the AvailabilityTaskExpanderHTNMethod and RouteReplacerHTNMethod subclasses, which are described in the Method Dictionary section.

6. Risk Interval Data Structures

The prototype uses the same four tiers of risk objects as described in Chapter IV. Risk sets, risk intervals, and risk subregions each have a corresponding software class: RiskIntervalSet, RiskInterval, and RiskIntervalSubregion, respectively. Risk segments are represented in the internal definitions of RiskIntervalSubregion.

A complication would arise if we tried to build RiskIntervals from AvailabilityTasks, during which units simply do nothing in some location. The approach of the prototype is to ignore the “risk” of these ephemeral tasks, since they will mostly be replaced with FireSupportTasks. We would not want to add a FireSupportTask to the PartialPlan to deal with an AvailabilityTask’s RiskInterval, only to have that AvailabilityTask later disappear. A similar issue for replaced routes (as described in

Section D.2 of Chapter IV) is less impactful because part of the new route is often still supported by the existing tasks.

RiskIntervalSets are used during planning (by FireSupportPlanner), but they are not recognized or used by the CSE. Therefore, RiskIntervalSets are stored in TaskNodes. On the other hand, the procedure to create a RiskIntervalSet for an OperationalTask is dependent on the type of task and its parameters. Therefore, OperationalTask has a BuildRiskIntervals() member function to construct and return (but not store) a RiskIntervalSet. This is easiest to do inside the OperationalTask class, with access to all internal parameter values. If we were not able to modify the OperationalTask source code (such as when adding a planning system to an existing CSE), we would need to create a helper function to build risk intervals for each OperationalTask type that the planning approach requires.

The default BuildRiskIntervals() function defined in the abstract OperationalTask class simply returns an empty set. In this implementation, only OperationalActions can build a nonempty RiskIntervalSet (by overriding BuildRiskIntervals()). The RiskIntervalSet of a TaskNode<T> where T is not an OperationalAction is the union of the RiskIntervalSets in TaskNodes below it in the PartialPlan.

a. Risk Value Approach

For this prototype, we use the basic $K = I$ approach (as explained in Section I.4 of Chapter IV) for threat unit target selection. Each RiskInterval's base risk value is totaled with the assumption that the threat unit will direct all of its fire at the friendly unit named in that RiskInterval, regardless of other simultaneous potential targets.

We do not use the precise locations of each entity in the threat unit for risk value calculations (even though we do for pathfinding costs during graph annotation). Instead, we use the CentroidPosition of the threat unit (the average of its entity's location coordinates) as the assumed location of all of its members. Additionally, we assume that all threat unit members can target all of the friendly Unit members for the whole duration of the RiskInterval, even if only a subset of the threat entities were in the IAnnotatedMobilityNodes' Threats annotation. These simplifications improve efficiency

in a few different ways. Without them, if a friendly unit's path went through several nodes with slightly different threat entity sets (from the same threat unit), then we would need to calculate a component of the risk value for each entity at each coordinate in each RiskIntervalSubregion's Waypoints (which store risk segment information; see Appendix B). Our single-location approach lets us perform just one calculation per Waypoint and multiply that number by the (assumed) constant number of entities in the threat unit. This results in a generally overestimating approximation because it counts expected losses from some threats that will not be able to target the friendly unit. This "cautious" approach falls in line with the other implementation components, and it provides some protection against the approximating threat annotation method (see the Annotated Mobility Graph section above). Although threat (defending) entities do not move in this prototype, this approach would still be effective against small-scale threat repositioning, as long as the threat Unit's CentroidPosition remains in the same approximate location.

The prototype uses risk reduction coefficients, as described in Section H.3 of Chapter IV, with a fixed $\beta = 0.1$. This means that each suppressing unit is assumed to reduce the killing rate of the target to a tenth of its previous amount. This constant is set at compile time in the SUPPRESSION_COEFFICIENT field of the SupportByFire OperationalAction. We choose this value based on experiments with the WXXI suppression model in which fully suppressed entities (with available targets) had their rate of fire reduced to 0.03 of the unsuppressed rate. The Unit's killing rate also depends on the fraction of time that its members spend in a suppressed state—in some cases, entities can switch back and forth between different states while subjected to marginally effective suppressive fire. This is uncommon but possible for some longer-range fire support tasks. The value 0.1 is usually a good overestimate of threat unit effectiveness while suppressed, while still providing a significant drop in residual risk value to allow the subsequent selection of other suppression targets by the planning algorithm.

b. RiskIntervalSet

The RiskIntervalSet class adds some special features to a HashSet of RiskInterval objects. Since we often need to iterate through a subset of a RiskIntervalSet matching a

specific attacking unit or defending unit, the class maintains hash tables (of type Dictionary<Unit, ICollection<RiskInterval>>) for each of these subsets. These are accessed by the RisksFromThreatUnit(Unit) and RisksForFriendlyUnit(Unit) member functions. RiskIntervalSet provides the set operations of addition, removal, copying, cardinality, and several utility functions, which are listed in Appendix B.

c. RiskInterval

This class is an implementation of the theoretical concept of the same name. Each RiskInterval represents an uninterrupted interval of time where a single defending unit threatens (can target and damage) a single attacking unit. Its internal representation consists of a few primitive values and class references plus a linked list of RiskIntervalSubregion objects.

d. RiskIntervalSubregion

Each RiskIntervalSubregion represents its contained risk segments with lists of waypoints and times. The details of the base risk value calculation for risk segments are actually encapsulated in the TargetHandler class, which is described below. However, RiskIntervalSubregion stores several reusable intermediate values for this calculation: WaypointTimes, SegmentLength, ThreatDistance, ThreatDistanceSqr, MidPointThreatDistance, SpeedSqr, and ThreatDirectionDotVelocityTimes2 (see Appendix B).

e. RangeCrossingPoints

The current version of WXXI allows piecewise-defined probability of hit functions. Most combat models require at least two pieces for such a function, since targets beyond the maximum range are usually given a fixed hit probability of 0. The two Weapon types of the prototype (M16A2Weapon and M240MachineGun) are defined with four pieces: short, medium, and long range, and the remaining piece outside the maximum effective range. Short range only applies to moving shooters, so it is not relevant to the static threats of the prototype. Both of the prototype's Weapons have a linearly decaying probability of hit, with different rates of decay at medium and at long

range. Therefore, there are two points of concern for calculating base risk values: the medium range crossing point (the point where the rate of decay changes) and the maximum range.

`RangeCrossingPoints(Vector3 startPoint, Vector3 endPoint, float duration, Vector3 firingPoint, out List<float> crossingTimes)` is located in the `ThreatHandler` class to support possible future implementations with multiple Weapons per entity. It also provides a single access point for all targeting-related data. `ThreatHandler` accesses the `Weapon` object of its entity for specific `Weapon`-related data, acting as a point of indirection between the `RiskInterval` class and `Weapon` subclasses. The function uses the `firingPoint` parameter as the source of its calculations rather than the current location of its entity, which allows calculation from the assumed unit position. It provides a list of coordinates as its return value, and a corresponding list of crossing times in its output parameter. These are the points and times at which the moving target will cross from one piece of the `Weapon`'s probability of hit function to another. Both locations and time points are needed to construct a `RiskIntervalSubregion`.

`RangeCrossingPoints()` calls `RangeCrossingTimes(Vector3 velocity, float duration, float startDirectionDotVelocity, float startRangeSqr, float maxRangeSqr)` to solve the quadratic distance equation with respect to time. It then uses the times to linearly interpolate the crossing points between `startPoint` and `endPoint`, assuming constant speed.

`RangeCrossingPoints()` only determines the crossing points; it does not label the segments with their correct probability of hit equation. In a later processing step, `RiskValueLinearIntegral()` makes that determination by testing the range to segment midpoints. In the case of a tangential range point intersection—that is, a route segment that touches a range endpoint but does not cross into a different piece of the probability of hit equation—it is possible that the midpoint test could land on the wrong side of the range point due to floating point error. Therefore, `RangeCrossingTimes()` adds an additional “crossing point” (actually a tangent point) if it detects a tangential intersection. This causes the midpoints of the two resulting segments to be some distance from the tangent point, reducing the chances of an erroneous midpoint test.

f. RiskValueLinearIntegral

This implementation uses the precise integral-based approach to computing base risk values for a risk segment. It is a direct implementation of the base risk value calculation derived in Chapter IV, Section H.2. The resulting equations from that section, after a few algebraic manipulations, are reprinted here:

$$\Psi(z_{i,j}) = \alpha \left(a_{\max} \tau'' - \int_0^{\tau''} D_x(\mathbf{y}, \tau) d\tau \right),$$

$$\int_0^{\tau''} D_x(\mathbf{y}, \tau) d\tau = \frac{1}{2ss^2} \left((a's + b)(a's - b) \ln \frac{|a''s + s^2\tau'' + b|}{|a's + b|} + (s^2\tau'' + b)a''s - a'sb \right)$$

The terms of these equations map to the variables of RiskValueLinearIntegral as follows:

- τ'' : duration
- s, s^2 : speed, speedSqr
- $s\tau''$: distance
- a', a'^2, a'' : startRange, startRangeSqr, endRange
- b : b

RiskValueLinearIntegral stores the result of $\int_0^{\tau''} D_x(\mathbf{y}, \tau) d\tau$ as distanceIntegral, then computes $\Psi(z_{i,j})$ by calling the getExpectedKills() function of the appropriate Weapon object (see below). To implement the formula for distanceIntegral in code, we introduce some additional variables for repeated expressions:

$$\begin{aligned} c0 &= a's \\ c1 &= a's + b \\ c2 &= a''s \\ c3 &= s^2\tau'' + b \end{aligned}$$

The following version of the formula is computed in RiskValueLinearIntegral(). Square brackets show where expressions are replaced with the “c” variables.

$$\begin{aligned}
& \int_0^{\tau''} D_x(\mathbf{y}, \tau) d\tau \\
&= \frac{1}{2ss^2} \left([a's + b]([a's] - b) \ln \frac{[a''s] + [s^2\tau'' + b]}{[a's + b]} + [s^2\tau'' + b][a''s] - [a's]b \right) \\
&= \left(c1(c0 - b) \ln \frac{|c2 + c3|}{|c1|} + c3 * c2 - c0 * b \right) / (2ss^2)
\end{aligned}$$

g. *getExpectedKills*

The Weapon class includes an abstract member function `getExpectedKills(float selectedRange, double rangeIntegral, float shooterSpeed, float targetSpeed, float protection, PlatformType platform, float duration)`, which is implemented in `DirectFireWeapon` (the superclass of both Weapon classes used in the prototype). This implementation takes advantage of `DirectFireWeapon`'s organization of combat resolution data and code. Most of the same properties and functions used during normal combat resolution are also used in `getExpectedKills()`. Since the base risk value (the expected kills) is the integral of the instantaneous expected killing rate, the variable part of the equation is separated from the constant part. The constant part is mostly contained in a function called `probHitCoefficient()`. During a simulation run, the probability of hit is calculated as `probHitCoefficient() * baseProbHit()`; but during planning, the expected number of hits is calculated as `probHitCoefficient() * baseProbHitIntegral()`. Both `baseProbHit()` and `baseProbHitIntegral()` have a few different calculation branches based on which piece of the probability-of-hit function the target resides in. To determine the expected kills from the expected hits, `getExpectedKills()` multiplies the expected hits by `ExpectedRateOfFire * getProbKillGivenHit()`.

The `probHitCoefficient(float range, float shooterSpeed, SuppressionState shooterState, float targetSpeed, float targetProtection)` function determines the number to multiply by the probability of hit to account for the shooter's movement, the target's movement, the suppression state of the shooter, and the protection of the target. Assuming constant velocities of shooter and target, these values remain constant as the range to the target changes, so they are the same for both the instantaneous hit rate and

the integral of the hit rate. For base risk value calculations, we assume the attacking unit is moving at normal traveling speed with no protection, and that the defender (the shooter) is not moving.

The `baseProbHit(float range, bool shooterMoving)` function determines the component of the probability of hit that depends on target range. The formula from the C# code, translated into mathematical notation, is

$$p_H(a) = \begin{cases} p_{Hm}(a) = P_{\max} - (P_{\max} - P_{\text{med}}) \frac{a}{a_{\text{med}}} & \text{if } a < a_{\text{med}} \\ p_{Hl}(a) = P_{\text{med}} \left(\frac{a_{\max} - a}{a_{\max} - a_{\text{med}}} \right) & \text{if } a_{\text{med}} \leq a \leq a_{\max} \\ 0 & \text{if } a > a_{\max} \end{cases}$$

where p_H is the probability of hit, p_{Hm} is the probability of hit at medium range, and p_{Hl} is the probability of hit at long range. The variables in these expressions represent the following variables and properties in C# code:

- $a \Leftrightarrow$ `range` or `selectedRange`: the range to the target at the time of the shot.
- $a_{\text{med}} \Leftrightarrow$ `MediumRange`: the target range at which the medium range rate of decay of the probability of hit changes to the long range rate of decay
- $a_{\max} \Leftrightarrow$ `MaxEffectiveRange`: the maximum effective range of the Weapon, beyond which the probability of hit is zero
- $P_{\text{med}} \Leftrightarrow$ `MediumRangeProbHit`: the base probability of hit when the target range is exactly a_{med}
- $P_{\max} \Leftrightarrow$ `BestProbHit`: the maximum possible base probability of hit (at point blank range)

To integrate this over time, we change a to a function of time $a(t)$. Using the distance integral I already computed by `RiskValueLinearIntegral`, we need only a little more derivation:

$$\int_0^{\tau''} p_{Hm} a(\tau) d\tau = P_{\max} \tau'' - (P_{\max} - P_{\text{med}}) \frac{I}{a_{\text{med}}},$$

$$\int_0^{\tau''} p_{Hl} a(\tau) d\tau = P_{\text{med}} \left(\frac{a_{\max} \tau'' - I}{a_{\max} - a_{\text{med}}} \right)$$

These equations are computed (with the C# variable names given above) by `baseProbHitIntegral()`.

7. Mission Planner

With the component federation prescribed by the Conceptual Planning Framework, we would normally expect each Mission Planner and Enhancement Planner to reside in a separate class, namespace, or package. In this prototype, the singular Mission Planner is just a function (`planManeuver()`) within the `FireSupportPlanner` class. This architecture is simply for expediency of development: objects such as the `PartialPlan` can be shared within the class context. The `planManeuver()` function's purpose is to generate a `PartialPlan` with the correct structure and reasonable data as input for the `PlanFireSupport()` function, which is described in the next section.

The `planManeuver()` function is not a true automated tactics engine because most of its decisions are made by the user. It is more correctly described as a translator from the Unity Scene View input into a `PartialPlan` with a military hierarchy. The details of `planManeuver()` are described in the Method Dictionary section (above) and Appendix B. We provide a more narrative summary here.

Regardless of the number of echelons of the Friendly Force, `planManeuver()` generates a `PartialPlan` with only the following types of platoon- and squad-level `TaskNodes`. Units above or below these echelons (for example, those whose names start with “FireTeam,” “Company,” or “Battalion”) are untasked, even if some of their subordinate units are tasked. For these descriptions, the term *maneuvering* means that a Unit's `IsManeuverTaskable` flag is set to true. The `planManeuver()` function does the following:

- Maneuvering infantry platoons are assigned a `TaskNode<AssaultTask>`. This `TaskNode`'s Children include a platoon movement to an assault

position followed by a TaskNode<MissionTask> named “Squad Assault” for each subordinate maneuvering rifle squad.

- Maneuvering rifle squads not subordinate to a maneuvering platoon are assigned a TaskNode<MissionTask> named “Squad Assault.”
- Non-maneuvering Units with IsFireSupportTaskable set to true are assigned a single TaskNode<AvailabilityTask> under the top-level TaskNode, which will later be used by PlanFireSupport().
- Maneuvering Units with IsFireSupportTaskable set to true are assigned a single TaskNode<AvailabilityTask> as a child of the Unit’s top-level TaskNode and successor to its latest-ordered sibling TaskNode.

The planManeuver() function uses the location of each maneuvering Unit’s InitialFormationLeader and the locations of the Assault Position and Objective GameObjects of the Unit’s WaypointRoute as starting and ending points for tactical pathfinding. Each of these points is set by the user in Unity’s Scene View and Inspector View. The only planning details not directly controlled by the user are the paths between points and the path traversal times (which are dependent on the paths’ waypointPaths and the Unit’s speed). This makes a planning cost function unnecessary, but the tactical pathfinding cost function affects unit paths between TacticalControlMeasures. An example of a company-sized maneuver plan is shown in Figure 39.

```

▼ wxxi.MissionTask 1 (Company) [1, 10000] (Seize)
  ▼ [AssaultTask 9: unit=Platoon (1) (wxxi.Unit), enemyUnit=, startTime=100, endTime=1726.899]
    wxxi.ChangeFormationAction 6 (Platoon (1)) [100, 100]
    wxxi.MoveByRouteAction 7 (Platoon (1)) [100, 1476.888]
    ▼ wxxi.MissionTask 15 (Rifle Squad v3 (1-1)) [1476.888, 1710.502] (Squad Assault)
      wxxi.ChangeFormationAction 11 (Rifle Squad v3 (1-1)) [1476.888, 1476.888]
      wxxi.MoveByRouteAction 12 (Rifle Squad v3 (1-1)) [1476.888, 1500.502]
      wxxi.ChangeFormationAction 13 (Rifle Squad v3 (1-1)) [1680.502, 1710.502]
      wxxi.AssaultAction 14 (Rifle Squad v3 (1-1) vs. EnemyFireTeam (1-1)) [1710.502, 2069.828]
      ► wxxi.AvailabilityTask 2488 (Rifle Squad v3 (1-1)) [2069.828, 10000]
    ▼ wxxi.MissionTask 22 (Rifle Squad v3 (1-2)) [1476.888, 1726.899] (Squad Assault)
      wxxi.ChangeFormationAction 18 (Rifle Squad v3 (1-2)) [1476.888, 1476.888]
      wxxi.MoveByRouteAction 19 (Rifle Squad v3 (1-2)) [1476.888, 1516.899]
      wxxi.ChangeFormationAction 20 (Rifle Squad v3 (1-2)) [1696.899, 1726.899]
      wxxi.AssaultAction 21 (Rifle Squad v3 (1-2) vs. EnemyFireTeam (1-3)) [1726.899, 2273.451]
      wxxi.AvailabilityTask 23 (Rifle Squad v3 (1-2)) [2273.451, 10000]
    ▼ wxxi.MissionTask 29 (Rifle Squad v3 (1-3)) [1476.888, 1716.95] (Squad Assault)
      wxxi.ChangeFormationAction 25 (Rifle Squad v3 (1-3)) [1476.888, 1476.888]
      wxxi.MoveByRouteAction 26 (Rifle Squad v3 (1-3)) [1476.888, 1506.95]
      wxxi.ChangeFormationAction 27 (Rifle Squad v3 (1-3)) [1686.95, 1716.95]
      wxxi.AssaultAction 28 (Rifle Squad v3 (1-3) vs. EnemyFireTeam (1-2)) [1716.95, 2161.307]
      wxxi.AvailabilityTask 30 (Rifle Squad v3 (1-3)) [2161.307, 10000]
  ► [AssaultTask 34: unit=Platoon (2) (wxxi.Unit), enemyUnit=, startTime=100, endTime=2426.675]
  ▼ wxxi.MissionTask 53 (Rifle Squad v3 (2-3)) [100, 871.2499] (Rifle Squad Assault)
    wxxi.ChangeFormationAction 49 (Rifle Squad v3 (2-3)) [100, 100]
    wxxi.MoveByRouteAction 50 (Rifle Squad v3 (2-3)) [100, 841.2499]
    wxxi.ChangeFormationAction 51 (Rifle Squad v3 (2-3)) [841.2499, 871.2499]
    wxxi.AssaultAction 52 (Rifle Squad v3 (2-3) vs. EnemyOP (3)) [871.2499, 1098.04]
    ► wxxi.AvailabilityTask 3556 (Rifle Squad v3 (2-3)) [1098.04, 10000]
  ► [AssaultTask 58: unit=Platoon (3) (wxxi.Unit), enemyUnit=, startTime=100, endTime=1261.645]
  ► wxxi.AvailabilityTask 2521 (MachineGun Squad v3 (1)) [1, 10000]
  ► wxxi.AvailabilityTask 3093 (MachineGun Squad v3 (2)) [1, 10000]
  ► wxxi.AvailabilityTask 2815 (MachineGun Squad v3 (3)) [1, 10000]

```

The AssaultTask for Platoon (1) is shown in full detail here, except for its AvailabilityTasks. Its squad-level assault plans are MissionTasks 15, 22, and 29. MissionTask 53 is the plan for an independent squad that does not fall under the command of a platoon.

Figure 39. A Company Maneuver Plan Displayed in the Unity Transform Hierarchy

8. Enhancement Planner

The PlanFireSupport() function of the FireSupportPlanner class is the singular Enhancement Planner of the prototype. In is an implementation of the Fire Support Planner described in Chapter IV.

The objective function for the fire support planner is the residual risk value of the PartialPlan, as described in Chapter IV. The cost of the current plan is determined by its RiskIntervalSet and accessed through the property PartialPlan.RiskValue. The score of a potential fire support task is determined by its provisional reduction in risk to the current plan, and accessed through the property FireSupportTask.RiskDelta.

PlanFireSupport() uses a few helper functions and supporting classes, which align with subroutines and concepts introduced in Chapter IV. Some of these are explained in the Risk Interval Data Structures section above. We proceed now with descriptions of the other supporting classes, followed by the helper functions, and conclude with a narrative description of PlanFireSupport() itself.

a. Availability Sets

The AvailabilitySet holds all of the non-decomposed TaskNodes in the PartialPlan whose contained OperationalTask is an AvailabilityTask. Only one AvailabilitySet, called availabilityTasks, is instantiated during fire support planning. It corresponds to the set A in Algorithm 1 (see Chapter IV). The primary purpose of AvailabilitySet is to perform set operations on TaskNode<AvailabilityTask> objects.

A unique functionality of AvailabilitySet is *route caching*, which works as follows. Whenever a FireSupportPositionFinder is used to find paths, the starting point (for all paths) comes from a single AvailabilityTask. The FireSupportPositionFinder generates paths to enough positions to target all possible threats. If we need to create new potential FireSupportTasks from that AvailabilityTask in a later planning iteration, the path results would be exactly the same. To avoid rework, every time an IAnnotatedMobilityPath is retrieved from FireSupportPositionFinder's UnitTargetingPaths to target a threat Unit, a mapping is cached with AvailabilitySet's StoreRoute(AvailabilityTask, Unit, IAnnotatedMobilityPath) function. This mapping is from the pair (a, e) to p , where a is an AvailabilityTask, e is a threat Unit, and p is an IAnnotatedMobilityPath. The IAnnotatedMobilityPath can later be retrieved by the RouteLookup(AvailabilityTask, Unit) function.

Similarly, whenever a point-to-point route is found for a particular unit template, there is no need for another pathfinding search between that start and end point. We cache a mapping from the triple $(u, \mathbf{c}, \mathbf{c}')$ to π , where u is an EntityInformation object (our prototype's stand-in for a unit template), \mathbf{c} is a start point for a path, \mathbf{c}' is its end point, and π is an IAnnotatedMobilityPath. The mapping is created by StoreRoute(EntityInformation, Vector3, IAnnotatedMobilityPath). Note that the function signature differs from the version of StoreRoute() mentioned in the previous paragraph (StoreRoute has a multi-target path version and a point-to-point version). The end point (\mathbf{c}') is extracted from the IAnnotatedMobilityPath. Cached point-to-point paths are retrieved with the TryRouteLookup(EntityInformation, Vector3, Vector3, out IAnnotatedMobilityPath) function, which returns a Boolean indicating whether any such point-to-point path was in the cache. If true, the IAnnotatedMobilityPath output parameter will have a reference to the retrieved path.

Placing route-caching functionality in AvailabilitySet is an expedience for prototype development, but this capability is almost certainly useful for other Mission Planners and Enhancement Planners. A more mature implementation would place route caching in the Pathfinding component, separate from AvailabilitySet. Having transparent caching operations would simplify the code of multiple Mission Planners and Enhancement Planners that depend on them.

b. Potential Fire Support Sets

Representing the set P of Algorithm 1, a PotentialFireSupportSet holds all of the TaskNode<FireSupportTask> objects that are not part of the PartialPlan but being considered for addition to it. Only one PotentialFireSupportSet, called potentialTasks, is maintained by FireSupportPlanner.

Each addition to a PotentialFireSupportSet requires both a set (or singleton) of TaskNode<FireSupportTask> objects and the TaskNode<AvailabilityTask> from which they were (or it was) generated. A mapping between these is stored, allowing the future determination of the GeneratingTask, as seen in the ApplyTask pseudocode of Chapter IV, Section F.3.

When a TaskNode<FireSupportTask> is removed from a PotentialFireSupportSet, its associated TaskNode<AvailabilityTask>, and all other TaskNode<FireSupportTask> objects generated from the same TaskNode<AvailabilityTask>, are automatically removed as well. This step corresponds to the subtraction of $\text{GenerateTasks}(a_w, R, R, W)$ from P in the ApplyTask pseudocode.

Other properties and functions provided by PotentialFireSupportSet are listed in Appendix B.

c. *Fire Support Tasks*

FireSupportTask is an implementation of the mathematical object of the same name from Chapter IV, defined there as $w = (f, \pi, \mathbf{c}, \phi, e, t_1, t_2, t_3)$. The parameters of FireSupportTask (including generic parameters inherited from OperationalTask), with their corresponding mathematical attributes, are

- Unit (inherited): f , the fire support unit
- EnemyUnit (inherited): e , the targeted threat
- StartTime (inherited): t_1
- EffectsTime: t_2
- EndTime (inherited): t_3
- RouteToPosition: ρ , which is left as null if InitialLocation = SupportByFireLocation
- InitialLocation: the starting coordinates of RouteToPosition
- SupportByFireLocation: \mathbf{c} , the center coordinates of the firing position
- SupportByFireGraphNode: the IAnnotatedMobilityNode containing SupportByFireLocation
- NextFireSupportTask: $w' = \varpi(W, a)$, where W is the current PartialPlan and a is the GeneratingTask of w . This is a reference to the next FireSupportTask for f occurring in the PartialPlan for which this FireSupportTask is being considered (OperationalTasks do not normally have references to other OperationalTasks, but this is a special case).

- RouteToNextPosition: the route $\rho_{\mathcal{U}}$ of $w_{\mathcal{U}} = \mathcal{P}(w, w_{\mathcal{U}})$. When $\varpi(W, a)$ is undefined, RouteToNextPosition is set to null. A new FireSupportTask corresponding to $w_{\mathcal{U}}$ is not actually constructed until definitely needed.

Each FireSupportTask can report its current score with the RiskDelta property. The score is determined by comparing affected risk sets. To handle these, FireSupportTask breaks the normal rule (of this implementation) that RiskIntervals are stored in TaskNodes rather than OperationalTasks. This approach is taken for efficiency and to maintain separation of task-specific logic from the TaskNode class, which might later be replaced by a different partial plan implementation. The RiskIntervals stored by FireSupportTask correspond to the sets $R_{\Delta}, R'_{\Delta}, R_{\nu}$ defined in Chapter IV:

- (1) IgnoredRiskIntervals: references to the RiskIntervals that would be removed from the PartialPlan (along with RouteToNextPosition) if this FireSupportTask were added to the PartialPlan. Each member of IgnoredRiskIntervals is a member of the current PartialPlan's RiskIntervalSet.
- (2) PriorAffectedRiskIntervals: references to the RiskIntervals that would be modified (by reduction of the RiskValue of a RiskIntervalSubregion) if this FireSupportTask were added to the PartialPlan. Each member of PriorAffectedRiskIntervals is a member of the current PartialPlan's RiskIntervalSet. The union of IgnoredRiskIntervals and PriorAffectedRiskIntervals corresponds to R_{Δ} .
- (3) PostAffectedRiskIntervals: copies of the PriorAffectedRiskIntervals that have been modified to reflect the provisional addition of this FireSupportTask to the PartialPlan. Members of PostAffectedRiskIntervals are not members of the current PartialPlan's RiskIntervalSet. PostAffectedRiskIntervals corresponds to $R'_{\Delta} - R_{\nu}$.
- (4) AdditionalAffectedRiskIntervals: R_{η} , the new RiskIntervals that will be added to the PartialPlan if this FireSupportTask is added to the PartialPlan. These RiskIntervals are a result of threats to the tasked Unit, either during the subordinate SupportByFire task or one of the FireSupportMoveByRoute tasks. Members of AdditionalAffectedRiskIntervals are not (yet) members of the current PartialPlan's RiskIntervalSet.

Each member of PostAffectedRiskIntervals is created by copying a member of PriorAffectedRiskIntervals, then using the ReduceRisk() function to lower its RiskValue.

FireSupportTask stores a bidirectional hash-map between each member of PriorAffectedRiskIntervals and its possible future version in PostAffectedRiskIntervals. Utility functions of FireSupportTask are listed in Appendix B.

d. NextFireSupportTaskNode Function

As described in Chapter IV (see the Planning Operators section), the Fire Support Planner needs to be able to determine the next FireSupportTask occurring after a given AvailabilityTask a —that is, $\varpi(W, a)$. NextFireSupportTaskNode(AvailabilityTask) fulfills this requirement by taking advantage of the HasRequiredStartLocation property, which is always set to true on a SupportByFireAction.

The HasRequiredStartLocation property is meant to allow the FireSupportPlanner to insert FireSupportTasks *prior to* maneuver tasks, not just after them as the current version allows. A relatively minor extension to FireSupportPlanner would allow this, but testing it would require improvements to the maneuver planner. Currently, planManeuver() fulfills all maneuver tasks as quickly as possible, so there would be little time available prior to any maneuver tasks for insertion of fire support tasks.

e. GenerateTask Function

An implementation of the TryGenerateTask pseudocode (Section F.5 of Chapter IV), this function takes a TaskNode<AvailabilityTask>, a RiskInterval, the path to a position (from FireSupportPositionFinder), and a reference to the next FireSupportTask in the PartialPlan (found by NextFireSupportTaskNode()), and returns a TaskNode<FireSupportTask> object of maximal duration given those input constraints. If it cannot construct a valid FireSupportTask, it returns null. This function is relatively straightforward, its main purpose being separation of the mechanics of task construction from GenerateTaskOptions().

GenerateTask() gets an IAnnotatedMobilityPath to the SupportByFireLocation of the next FireSupportTask (if there is one) by either finding one in the cache (TryRouteLookup()) or invoking the Pathfinding module (FindRoute()). It uses the

LeadTraceTravelTime of this route to calculate the required departure time from the new (earlier) task's SupportByFireLocation.

The time point variables of GenerateTask() correspond to those of Algorithm 3. For example, τ_a matches t_a and τ_0 matches t_0 . The time calculations are identical to those of TryGenerateTask except that τ_a is set earlier by maxWarmupTime seconds (see Derived Combat Effects).

The new TaskNode<FireSupportTask> is decomposed using FireSupportDecompositionMethod. This adds the TaskNode<OperationalAction> Children necessary to build the RiskIntervals of the new FireSupportTask. These TaskNodes include FireSupportMoveByRouteActions for the path to the new task's SupportByFireLocation as well as the path to the next FireSupportTask's SupportByFireLocation (if applicable).

The RiskIntervals relevant for the new FireSupportTask's score need to be provided to the FireSupportTask object. GenerateTask() finds the IgnoredRiskIntervals in the next FireSupportTask—these come from the path-related tasks that would be removed if the new FireSupportTask were added to the plan. Its AdditionalAffectedRiskIntervals are all of those stored in its TaskNode's children. The PriorAffectedRiskIntervals and PostAffectedRiskIntervals come from the RiskIntervals in the PartialPlan; these are determined when GenerateTask() calls StoreEffectsOn() with each of the new TaskNode's Children.

f. GenerateTaskOptions Function

This function corresponds to the pseudocode of Chapter IV, Section F.4. Its only input parameter is a TaskNode<AvailabilityTask>, but it has access to the PartialPlan and its RiskIntervals. It creates a new FireSupportPositionFinder and then iterates through each of the <Unit, IAnnotatedMobilityPath> pairs in its UnitTargetingPaths.

Within each of these iterations, GenerateTaskOptions() examines all of the RiskIntervals resulting from the current threat unit—that is, the unit part of the

UnitTargetingPaths pair currently being investigated. It calls GenerateTask() to attempt to create a FireSupportTask against each of these RiskIntervals.

Each of the TaskNode<FireSupportTask> objects successfully created by GenerateTask() calls is returned as a TaskNodeSet<FireSupportTask>. None of these TaskNodes has been added to the PartialPlan yet; they are only potential tasks at this point.

g. PlanFireSupport Function

The core functionality of FireSupportPlanner is contained in the PlanFireSupport(double) function, which takes the target residual risk score as its only parameter. It is an implementation of the top-level greedy best-first search algorithm of Chapter IV. An implicit parameter is the maneuver plan, which must already be stored in the member field called plan (of type PartialPlan). As in the pseudocode, the two major phases of PlanFireSupport are the initialization and the planning loop. Preprocessing—i.e., preparation of the Annotated Mobility Graph—is not part of FireSupportPlanner because its results are needed by the maneuver planner, which runs earlier.

To initialize, the function finds all TaskNode<AvailabilityTask> objects in the PartialPlan, builds the PartialPlan’s RiskIntervals, and uses each OperationalAction in the PartialPlan to reduce the residual risk of each RiskInterval. The purpose of the last step is to ensure that all attrition effects from AssaultActions are recorded, preventing fire support units from targeting threats after they have been assaulted. Initialization concludes with a GenerateTaskOptions() call against every AvailabilityTask in the maneuver plan. This creates the initial pool of FireSupportTasks to choose from, all of which are stored in the potentialTasks member field.

The planning loop is written as a **for** loop to keep track of the iteration number. It terminates if any of the following are true at the end of an iteration:

- The PartialPlan’s RiskValue is less than or equal to targetScore—which never occurs in this prototype because we only call the function with a fixed argument of 0.0

- The number of iterations has exceeded `maxProcessingIterations`, which is set in the Unity Inspector window
- The expected total processing time after the next iteration, based on the previous iteration's duration, would be greater than `maxProcessingTime`
- There are no planning branches left to choose (`potentialTasks` is empty)

In each iteration, `PlanFireSupport()` first removes `selectedTask`, the member of `potentialTasks` with the best score. The corresponding `TaskNode<AvailabilityTask>` is also removed from `availabilityTasks`, and the sibling `TaskNode<FireSupportTask>` entries—those potential tasks generated from the same `AvailabilityTask`—are removed (and discarded) as a side effect. `PlanFireSupport()` then calls `UpdateTaskOptions` on `potentialTasks` to cause all remaining entries to update themselves according the selection of `selectedTask`. After this, it calls `ReduceRisk` to modify the `PartialPlan`'s `RiskIntervals` for the effects of `selectedTask`.

The next steps actually add the chosen task to the `PartialPlan`. First, the `RouteReplacerHTNMethod` pulls the new movement-related child `TaskNodes` from the new parent task. Then the `AvailabilityTaskExpander` creates the new, shorter `AvailabilityTasks`. `DecomposeSwap()` is used to add these, along with the `TaskNode<FireSupportTask>` object—the newly chosen task—to the `PartialPlan`. If there is an immediately following task, `RouteReplacerHTNMethod` pulls out its now-irrelevant movement tasks and replaces them with the ones it found in the chosen `TaskNode<FireSupportTask>`. Another `DecomposeSwap()` completes this step.

Now the function adds the new, shorter `TaskNode<AvailabilityTask>` objects to the `availabilityTasks` set and generates new potential tasks for them using `GenerateTaskOptions()`. It also generates a potential task against each new `RiskInterval` that was added to the `PartialPlan` in this iteration for each member of `availabilityTasks`.

The `PartialPlan` may be compiled into `BattlePlans` and executed after any iteration, or even after no iterations (the maneuver plan alone). This allows the planner to be cut short relatively quickly.

Since availabilityTasks and potentialTasks are member fields and can grow very large in the more complicated scenarios, PlanFireSupport sets them to null before exiting. This allows the .NET garbage collector to dispose of their contents when needed and avoids memory management problems during execution.

9. Plan Compiler

The BattlePlan class is the executable plan format of WXXI. Each BattlePlan is assigned to a single unit. The CompilePlans() function of the PartialPlan class creates a hash table of type Dictionary<Unit, BattlePlan>. It then iterates through all TaskNodes, extracting the OperationalTask of each and adding it to the tasked Unit's BattlePlan in the hash table. Although only the OperationalActions are needed in the BattlePlan, CompilePlans() adds all OperationalTasks (including "abstract" tasks that are not OperationalActions) to provide more debugging information. OperationalTask names still appear in the output console when they begin and end, even though they result in no additional activity in the simulation.

Since the prototype is purpose-built for implementing the Conceptual Planning Framework and the Fire Support Planner, we could have used PartialPlan (rather than BattlePlan) as the executable format. But HTNs (with the features described above) are not universal to production CSEs, so compiling from a PartialPlan to BattlePlans is meant to demonstrate how a generic CSE could be upgraded with an automated battle planning system with little impact to the existing code base. It also provides a concrete example of the Plan Compiler. "Compilation" here is made simple by the containment of OperationalTask objects with fully bound parameters in the TaskNodes. Each OperationalTask in the PartialPlan is already in the correct format and has the right data for execution.

10. Separate Planning Systems

The prototype planning system is structured in such a way that a separate planning systems implementation is possible, though not without some additional work. Nothing prevents multiple IOperationalPlanner-derived objects from being attached to multiple different units, but there are no tasks or methods currently written to allow one

planner to pass tasks to another. This approach would also require an ordering of planner execution—which is not currently in place—since higher-echelon planners would need to execute before their subordinate planners. As discussed in Chapter III, the practical usefulness of a multi-echelon implementation of separate planning systems would require derived models to estimate the decisions and results of lower echelons.

E. SUMMARY

We have presented a prototype implementation of the Conceptual Planning Framework containing, as its sole Enhancement Planner, a fire support planner based on the data structures, techniques, and algorithms discussed in Chapter IV. The implementation provides a concrete example of how the framework's organizational principles can be applied, and the development effort reciprocally informed the ideas of the framework. The fire support planner development likewise supported the underlying theory, and it allows us to present empirical evidence of its effectiveness and efficiency. This is covered in the next chapter.

VI. TESTING

A. OVERALL TEST DESIGN

We evaluate the correctness and effectiveness of the implementation with a three-step process, consisting of *functional*, *quantitative*, and *qualitative* testing. The maneuver planner and fire support planner are both subjected to functional testing. The quantitative and qualitative tests are focused on the fire support planner because its design and implementation are more novel and thorough than the maneuver planner.

In the remainder of this section, we explain the testing methodology in more detail and describe it in terms of verification and validation. We then cover the functional testing approach and present some of the results and lessons learned from functional testing. Both the quantitative and qualitative tests use a common set of scenarios, so we describe the scenario anatomy next. In the final two sections, we present the quantitative and qualitative results.

1. Iterative Testing and Development Process

The three testing phases are naturally described as sequential, but in practice, they form an iterative cycle that is connected to modeling development. Figure 40 illustrates how feedback loops from each phase can restart prior phases. We describe the testing process in terms of the inputs and outputs to each phase.

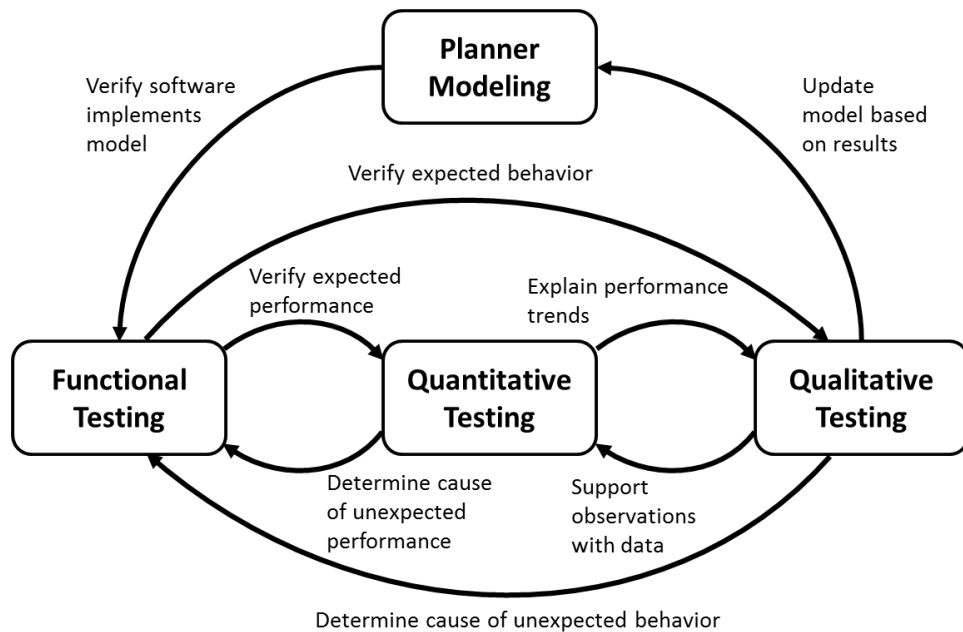


Figure 40. Testing Phases and Feedback Loops

a. Planner Modeling

The Planner Modeling phase is initiated first and is central to the testing and development process. Development of the conceptual model, the executable model, and the testing scenarios is all part of the Planner Modeling phase. Developing quantitative and qualitative test scenarios early in the development process provides valuable insight for software architecture decisions. This helps avoid the need for wholesale re-implementation late in development.

We put partially executable elements through functional testing as soon as possible to verify that the software behaves as intended. Of course, functional errors are cause for adjustments to the executable model. In some cases, though, apparent anomalies in the output actually identify missing aspects or errors in the planner’s conceptual model. Usually, unexpected quantitative results do not directly feed back to the modeling process—they first need to be understood for their functional or behavioral (qualitative) root causes.

b. Functional Testing

Functional testing is a combination of verifying desired computation and troubleshooting unexpected output. The former is input from modeling, and the latter is input from the other testing phases. Once components appear to function correctly in isolation, quantitative measurements and qualitative analysis provide new, more challenging stresses to the components.

c. Quantitative Testing

Quantitative measurements allow us to apply some statistical rigor to subjective and point observations about the planner's output. When a component appears to work correctly under functional testing, we test for positive or negative changes in the quantitative results. Generally, we expect an improvement; if we detect otherwise then we go back to functional testing to explain the anomaly. If all of the components appear to be functioning correctly, we subject the output to qualitative testing to find a tactical explanation for the statistical trends.

d. Qualitative Testing

In qualitative testing, we observe output plans—both statically and during execution—looking for resemblance to plans that humans might produce. When we see tactics that appear more or less realistic or effective, we use quantitative testing to back up those hypotheses. This combination of quantitative and qualitative checks helps sort out stochastic anomalies. When we observe unexpected tactical behavior that seems to agree with quantitative testing, we return to the modeling and functional testing steps in search of an explanation. These types of insights are presented with the qualitative results.

2. Considerations for Verification, Validation, and Accreditation

Modeling and simulation evaluation typically uses two types of tests: *verification* that the software “accurately represent the developer’s conceptual description and specifications” and *validation* that it is “an accurate representation of the real world from the perspective of the intended uses of the model” (DOD 2009a, 10). Our testing process

can support both verification and validation (V&V). Functional tests help verify that internal components are working according to specifications. Quantitative testing provides a “black box” approach to verification, permitting statistical checks of whether changes in inputs produce the expected changes in the outputs. If this kind of data is available from the simuland—for example, from real world experiments involving similar tactics on similar terrain—then quantitative testing also supports validation. Qualitative testing is primarily for validation, to the extent that we can compare properties and features of generated plans to those of real world plans. However, qualitative testing also supports verification since a qualitative anomaly may be the result of an implementation error.

Accreditation is “the official certification that a model or simulation and its associated data are acceptable for use for a specific purpose” (DOD 2009a, 9). Our work is focused on the technical and scientific aspects of verification and validation, and we do not seek “official certification” of WXXI or the ABPS implementation. Therefore, accreditation is out of scope.

V&V is typically described at the level of an entire simulation system. Note that, although our quantitative analysis depends on some overall combat performance measures, our testing process is only focused on components of the ABPS. Since we expect development of a planning system to proceed at a different pace and schedule than the development of its containing CSE, and since the performance of the planning system can be affected by the contents of the Planning Data, CSE and ABPS V&V should usually be done separately. However, ABPS V&V could be considered a component of a CSE V&V.

The running time of the planning system is a critical verification factor. The primary argument for having an ABPS is reduction of the end-to-end time required for an analytical study or training evolution. If the system is significantly faster than the legacy approach, where human operators must generate plans, then its adoption could result in more thorough and responsive output from M&S organizations, even if its combat performance results are no different from those resulting from human planning efforts. We cover computation time with the quantitative results.

B. FUNCTIONAL TESTING

In software engineering literature, functional testing evaluates a system against functional requirements without consideration of internal implementation decisions. Here, we use *functional testing* as more of a blanket term covering the following types of tests. These definitions are specific to this project.

- Unit testing: verifying that individual member functions produce the expected output in isolation from other software objects
- Component testing: verifying that major components (usually classes) of the automated planning system produce the expected output in isolation from other components
- Integration testing: verifying that components of the automated planning system perform as expected when connected to other components, including their external interfaces to the CSE (WXXI)

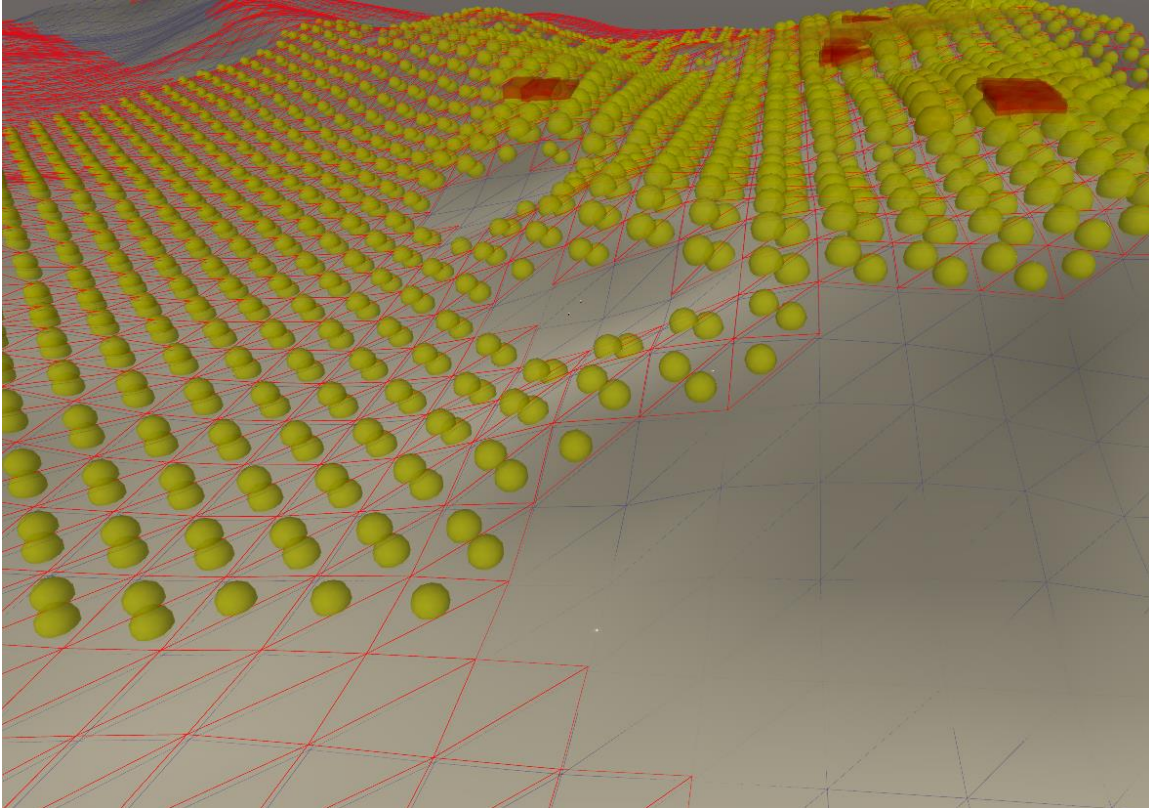
Since the approach of this project is rapid prototyping, we do not have an exhaustive testing plan progressing through all of the aforementioned test types for every module in the system. Instead, we present some key testing results as evidence that the system works as desired. Functional testing of this type has been performed throughout the development of WXXI and the ABPS.

Our functional testing makes use of the visual 3D tools available in the Unity environment to represent key aspects of the generated plan and its execution graphically. For elements that do not lend themselves to visual representation, or where a visual depiction would be too cluttered to be useful, we employ traditional textual debugging output using Unity's console logging system and debugging tools.

1. Annotated Mobility Graph Visualization

The Astar Pathfinding Project has built-in node visualization to support troubleshooting of navigation graphs. We have modified this visualization to display the upper and lower triangles of our terrain-based graph. Additionally, yellow spheres mark the center of each Annotated Mobility Node with a NodePenalty (risk value) greater than 0. Larger spheres represent greater penalties. The red triangles mark nodes that are visible by at least one Enemy Force entity, within a maximum range set by the user. Nodes

within sight of an Enemy Force entity but outside all Enemy Force entities' weapons range are marked with a red triangle but no yellow icon. These visualizations can be seen in Figure 41. (In place of the yellow icons, we can alternatively stamp the numerical risk value of each node onto the terrain.)



The lower triangles of each node are drawn in blue. The upper triangles of nodes that are visible by at least one Enemy Force entity are drawn in red. Yellow markers indicate nodes with nonzero penalty, meaning that we expect entities located in these nodes to be at risk of taking casualties. The red boxes highlight Enemy Force unit locations, which are the source of the risk values used to compute the node penalties.

Figure 41. Annotated Mobility Graph Visualization

The visualizations allow us to check that the nodes are generated in the correct locations and with the correct dimensions, that nodes threatened by defending entities are correctly identified, and that the killing rates (as input to the risk value calculations) are correctly decreasing with distance from the threats. The effectiveness of the approximate occlusion detection approach (i.e. testing the upper corners and centroid of each node) is

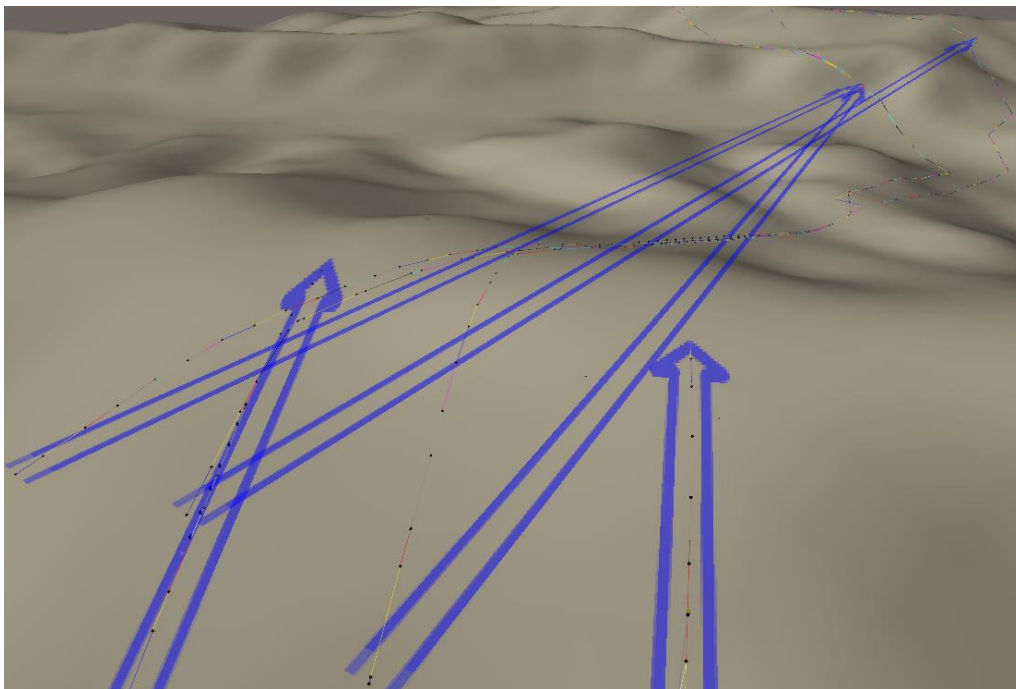
tested by placing Friendly Force entities in safe nodes (those with no yellow marker) inside the maximum weapons range of the Enemy Force. As expected, this results in no entities firing on the Friendly Force.

2. Geographic Task Visualization

Once a plan has been generated, we investigate its tactical decisions by displaying tasks symbolically in the 3D environment. MoveByRouteAction, FireSupportMoveByRouteAction, and AssaultAction tasks are displayed with the familiar “big blue arrow,” formally known as the axis of attack graphic (Figure 42). These represent a high-level abstraction, showing the tactical decisions that link different paths together. Detailed paths along the terrain can also be displayed (Figure 42 shows these, as well). These are useful for checking whether paths are chosen along tactically feasible routes and whether threatened nodes result in risk segments.

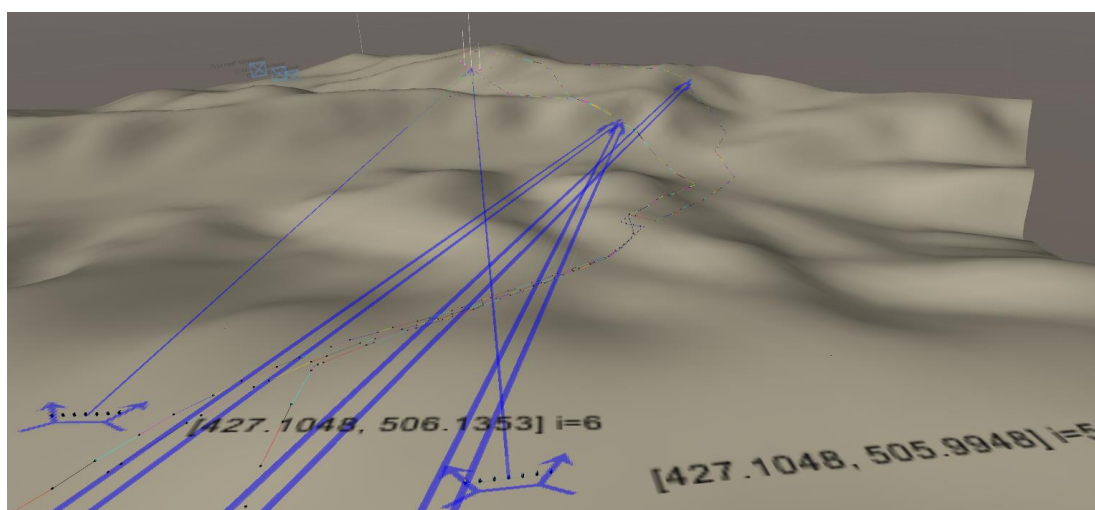
The support by fire position has its own doctrinal symbol, which we use to display SupportByFireAction tasks (Figure 43). These icons are codified in U.S. Army and Marine Corps doctrine (DOD 2010a, Ch. 7). Next to each SupportByFireAction symbol, we print a string of the form $[t_2, t_3]i = x$, where t_2 is the effects time, t_3 is the end time, and x is the number of the iteration during which the corresponding FireSupportTask was added to the PartialPlan.

For fire support tasks, we can check the sequence of blue arrows for each unit to determine whether route replacement is working correctly. The direction-of-fire arrows show whether a valid unit is being targeted. These icons also support qualitative analysis, where we go beyond validity to judge tactical planning decisions.



Wide blue arrows are drawn from the start point to the end point of movement tasks. The multi-colored line segments show the detailed paths that the formations are using to accomplish the movement tasks.

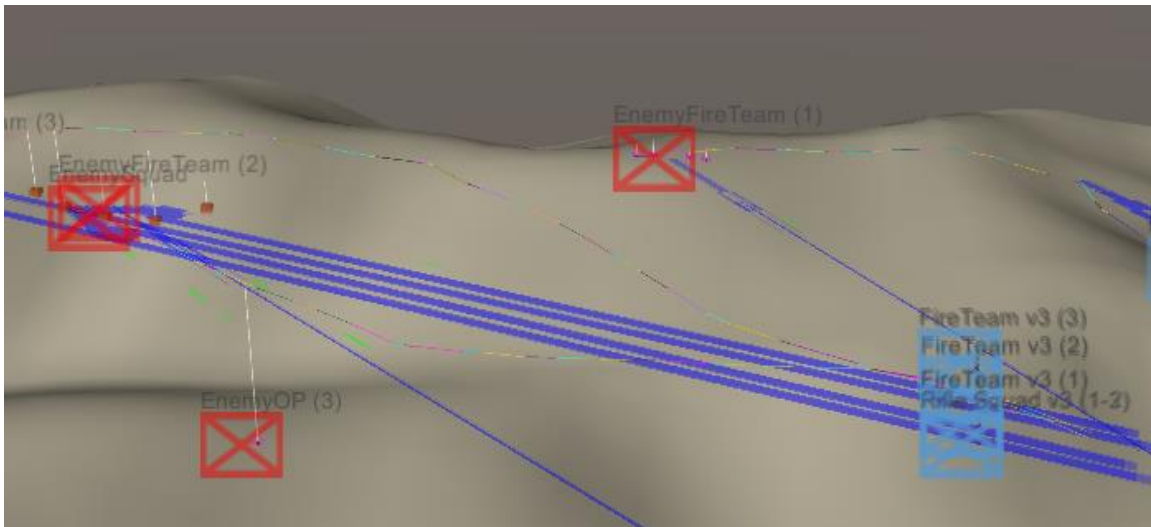
Figure 42. Movement Task Visualization



The blue symbols with two arrowheads represent support by fire tasks, and the thin blue lines point to the intended target unit. Simulation time and planning iteration number are printed next to the support by fire position.

Figure 43. Support By Fire Task Visualization

The user may pause WXXI prior to executing a replication, causing all tasks in the generated plan to be displayed simultaneously. This is useful to gain a general sense of how the plan will play out, but it can clutter the display. To examine the timing and individual tasks, the user can observe the replication at any desired speed relative to real-time (up to the processing limits of the hardware). During the replication, only the currently active tasks are displayed. We use this approach, along with the WXXI targeting, shot, and suppression visualization, to verify that Enemy Force units are being suppressed just prior to Friendly Force formations crossing into nodes with nonzero penalties (Figure 44).



The thin blue lines show which Enemy Force units have a SupportByFireAction directed against them. The white lines above the entities show the suppression weight currently applied to them. The green line segments represent individual shots or bursts.

Figure 44. Example of Suppression Effectiveness Evaluation

3. Hierarchical View of the Plan

To verify that output plans have the correct hierarchical and ordering structure, we retain the HTN task decomposition tree and display it in the Unity Hierarchy window. Figure 45 provides an example of this type of output. This hierarchical plan for a company is rooted at MissionTask 1. We first describe the maneuver part of the plan. Each platoon has an AssaultTask, and one independent squad has been tasked directly by

the company²⁵ (MissionTask 53). AssaultTask 9, assigned to the first platoon, has been partially expanded to show that it contains two platoon-level tasks (ChangeFormationAction 6 and MoveByRouteAction 12) and three squad-level tasks (MissionTasks 15, 22, and 29). The first of these squad tasks (MissionTask 15) has also been partially expanded.

Moving on to the fire support aspect of the plan, we note that FireSupportTasks are always children of AvailabilityTasks. The first one of these appearing in Figure 28 is AvailabilityTask 2488, appearing immediately after AssaultAction 14. This makes the first squad of the first platoon available to provide fire support after completing its assault. FireSupportTask 4013, a descendant of AvailabilityTask 2488, has been expanded to show that it consists of only one child task, a SupportByFireAction.

The company's machine gun squads each have an AvailabilityTask covering the entire duration of MissionTask 1. We have partially expanded some of the first machine gun team's availability tasks to show FireSupportTask 2539, which contains a ChangeFormationAction and a FireSupportMoveByRouteAction prior to its SupportByFireAction.

²⁵ This is a human decision made during maneuver plan development for a particular scenario.

```

▼ MissionTask 1 (Company) [1, 10000] (Seize)
  ▼ [AssaultTask 9: unit=Platoon (1) (Unit), enemyUnit=, startTime=100, endTime=1726.899]
    ChangeFormationAction 6 (Platoon (1)) [100, 100]
    MoveByRouteAction 7 (Platoon (1)) [100, 1476.888]
  ▼ MissionTask 15 (Rifle Squad v3 (1-1)) [1476.888, 1710.502] (Squad Assault)
    ChangeFormationAction 11 (Rifle Squad v3 (1-1)) [1476.888, 1476.888]
    MoveByRouteAction 12 (Rifle Squad v3 (1-1)) [1476.888, 1500.502]
    ChangeFormationAction 13 (Rifle Squad v3 (1-1)) [1680.502, 1710.502]
    AssaultAction 14 (Rifle Squad v3 (1-1) vs. EnemyFireTeam (1-1)) [1710.502, 2069.828]
  ▼ AvailabilityTask 2488 (Rifle Squad v3 (1-1)) [2069.828, 10000]
    ▼ AvailabilityTask 3991 (Rifle Squad v3 (1-1)) [2069.828, 2316.052]
      ► [FireSupportTask 3987: i=24, Unit=Rifle Squad v3 (1-1) (Unit), TargetUnit=EnemyFireTeamLast (2-1) (Unit), StartTime=2069.828, EndTime=2316.052]
    ▼ AvailabilityTask 4042 (Rifle Squad v3 (1-1)) [2230.428, 2316.052]
      ► [FireSupportTask 4013: i=26, Unit=Rifle Squad v3 (1-1) (Unit), TargetUnit=EnemyFireTeamLast (2-1) (Unit), StartTime=2230.428, EndTime=2316.052]
      ► [SupportByFireAction 4014: Unit=Rifle Squad v3 (1-1) (Unit), TargetUnit=EnemyFireTeamLast (2-1) (Unit), StartTime=2230.428, EndTime=2316.052]
      ► [FireSupportTask 4043: i=0, Unit=Rifle Squad v3 (1-1) (Unit), TargetUnit=EnemyFireTeamLast (2-1) (Unit), StartTime=2230.428, EndTime=2316.052]
    ► AvailabilityTask 2490 (Rifle Squad v3 (1-1)) [2604.41, 10000]
  ► MissionTask 22 (Rifle Squad v3 (1-2)) [1476.888, 1726.899] (Squad Assault)
  ► MissionTask 29 (Rifle Squad v3 (1-3)) [1476.888, 1716.95] (Squad Assault)
  ► [AssaultTask 34: unit=Platoon (2) (Unit), enemyUnit=, startTime=100, endTime=2426.675]
  ► MissionTask 53 (Rifle Squad v3 (2-3)) [100, 871.2499] (Rifle Squad Assault)
  ► [AssaultTask 58: unit=Platoon (3) (Unit), enemyUnit=, startTime=100, endTime=1261.645]
  ▼ AvailabilityTask 2521 (MachineGun Squad v3 (1)) [1, 10000]
    ▼ AvailabilityTask 3719 (MachineGun Squad v3 (1)) [1, 539.7773]
      AvailabilityTask 3720 (MachineGun Squad v3 (1)) [1, 190.8889]
    ▼ [FireSupportTask 2539: i=14, Unit=MachineGun Squad v3 (1) (Unit), TargetUnit=EnemyOP (3) (Unit), StartTime=190.8889, EndTime=539.7773]
      ChangeFormationAction 2541 (MachineGun Squad v3 (1)) [190.8889, 190.8889]
      FireSupportMoveByRouteAction 2542 (MachineGun Squad v3 (1)) [190.8889, 811.2499]
      [SupportByFireAction 2540: Unit=MachineGun Squad v3 (1) (Unit), TargetUnit=EnemyOP (3) (Unit), StartTime=811.2499, EndTime=811.2499]
    ► [FireSupportTask 3721: i=1, Unit=MachineGun Squad v3 (1) (Unit), TargetUnit=EnemyFireTeam (3-3) (Unit), StartTime=918.0, EndTime=918.0]
  ► AvailabilityTask 2754 (MachineGun Squad v3 (1)) [1550.712, 10000]
  ► AvailabilityTask 3093 (MachineGun Squad v3 (2)) [1, 10000]
  ► AvailabilityTask 2815 (MachineGun Squad v3 (3)) [1, 10000]

```

Some of the tasks in this plan have been collapsed for readability. The text of some of the tasks is cut off on the right hand side, but the structure of the plan is apparent without the missing details.

Figure 45. Partial Plan Hierarchical View

This method of viewing a plan supports verification that the HTN methods produce the desired structure. For example, although a FireSupportTask can carry a FireSupportMoveByRouteAction as a successor to its SupportByFireAction while in the PotentialTaskSet, this should be removed by RouteReplacementHTNMethod upon the addition of the FireSupportTask to the PartialPlan. This can be visually verified in the plan's hierarchical view by checking that, under each FireSupportTask, no FireSupportMoveByRouteAction appears after a SupportByFireAction.

4. Textual Display of Risk Intervals and Fire Support Tasks

One of the most challenging aspects of debugging the fire support planner is the large number of RiskInterval and FireSupportTask objects, especially those not yet added

to the plan. There can be many potential FireSupportTasks, and many of them cover the same geographical areas or units. Similarly, RiskIntervals can overlap in time and space, such as when two different Enemy Force units threaten the same Friendly Force unit at the same time. Graphically displaying all of these objects would result in unreadable clutter. A textual display is more helpful here.

The key pieces of information for these data objects are the units involved, the start and end times (along with the effects time for FireSupportTasks), and the score (RiskValue for RiskIntervals, RiskDelta for FireSupportTasks). Additionally, a *deterministic serial number* for each object is indispensable for determining object relationships, tracking down copy-by-value and copy-by-reference errors, and verifying the effects of code changes. By deterministic, we mean that each object receives the same serial number each time the planner is executed with the same input—which is not the case for built-in codes such as that returned by Unity’s GetInstanceID() function.

An example of textual output is provided in Figure 46. Here, we see that in iteration 14, the FireSupportTask with serial number 2539 has been selected to be added to the plan. The RiskInterval objects that it should be modifying (there are three in this case) are listed next. We can verify, for example, that these risk intervals’ times are overlapped by time interval [811.2499, 918.0398], which is the firing time of the FireSupportTask.

```

ITERATION 14 IN MAIN PLANNING LOOP OF PlanFireSupport
Total plan risk: 2470.7853597879

Chosen task: [FireSupportTask 2539: i=14, Unit=MachineGun
Squad v3 (1) (Unit), TargetUnit=EnemyOP (3) (Unit),
StartTime=190.8889, EffectsTime=811.2499, EndTime=918.0398,
InitialLocation=(2194.4, 611.8, 2918.6),
SupportByFireLocation=(1354.6, 701.6, 2075.6),
RouteToPosition=AnnotatedMobilityPath,
SupportByFireGraphNode=Pathfinding.TerrainTriangleMeshNode,
RiskDelta=153.43911716555, AffectedRiskIntervals.Count=3]

Affected risk intervals of chosen task;
RiskInterval 12354 (EnemyOP (3)) [829.3841, 841.2499]:
1.95719717354041;
RiskInterval 12355 (EnemyOP (3)) [841.2499, 871.2499]:
5.94578224504307;
RiskInterval 12356 (EnemyOP (3)) [871.2499, 918.0398]:
9.14581165986296;

```

Figure 46. Example of Textual Debugging Output

In some cases, it is helpful to copy the entire set of RiskIntervals or potential FireSupportTasks for two different iterations into a spreadsheet, where ordering and text matching tools can help identify which items have changed.

The fire support planner performs many calculations during a typical execution. Even if every individual calculation is implemented correctly, unintended values could still result from bookkeeping errors. To help detect this, we use a system of double-checks where possible. For example, Figure 47 is a code sample from the PlanFireSupport function. At the end of each iteration, this code verifies the D - score of the most recently added FireSupportTask by comparing its internal value with the actual difference in the plan's residual risk value. Small-magnitude discrepancies are ignored due to the possibility of floating point errors.

```

double priorPlanRisk = plan.RiskValue;
var selectedTaskScore = selectedTaskNode.Task.RiskDelta;
...
double actualRiskDelta = priorPlanRisk - plan.RiskValue;
double deltaDelta = System.Math.Abs (actualRiskDelta -
    selectedTaskScore);
if (deltaDelta > 0.00001) {
    Debug.LogError ("Erroneous RiskDelta in iteration "
        + fireSupportIteration);
}

```

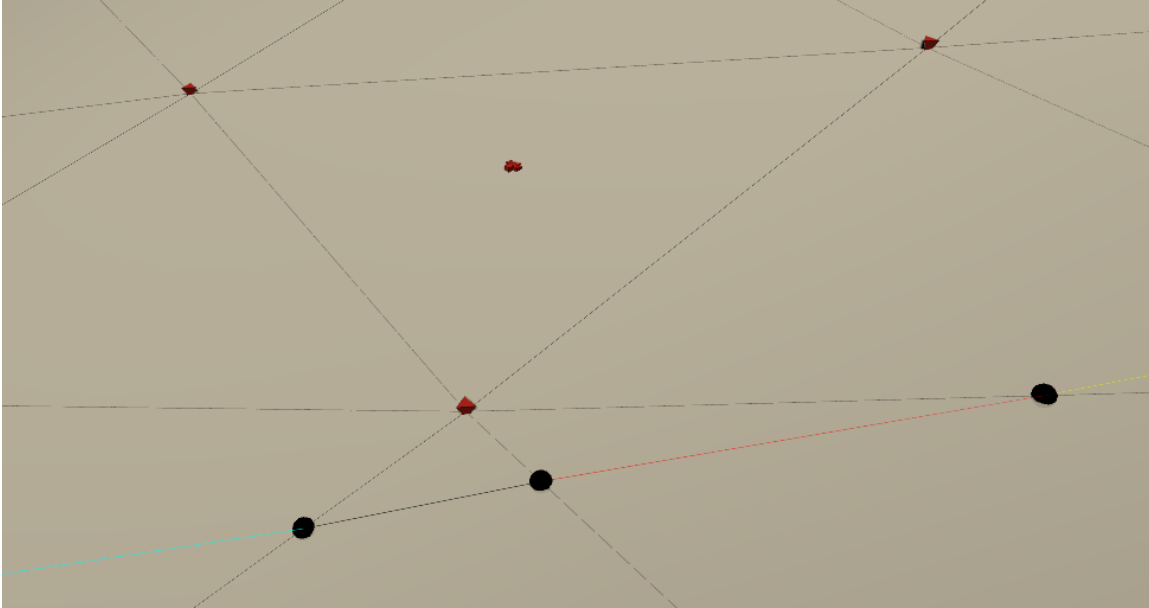
Figure 47. PlanFireSupport Score Verification Code Sample

These techniques are not particularly complicated, but without them, the complexity and volume of data involved with the fire support planner would make troubleshooting too time-consuming to be practical.

5. Geographic Error Visualization

When the above techniques are not sufficient, we catch errors and display the relevant geographic positions as 3D objects. This technique is useful in testing the functionality of new features, such as the subdivision of paths at each node boundary after modifiers have been applied. Figure 48 provides an example of this kind of troubleshooting. We can quickly see that the waypoints of a modified path do not take it through one of the path's nodes, an anomaly that could result in an incorrect RiskInterval construction depending on which Enemy Force units have visibility of the actual traversed nodes (this is a fictitious error for demonstration purposes).

This technique can be combined with the textual display technique for a powerful troubleshooting capability. The developer can use the textual methods to find the serial numbers of problematic objects and then insert code to place error markers (such as those in Figure 48) conditionally based on the serial numbers. Once the developer is able to visualize the problematic objects, the human eye tends to notice the out-of-place items, or at least provide clues for deeper investigation. For example, points very close to a corner or edge are suspect for floating-point or fixed-point rounding issues.



The route, marked with black markers and multi-colored line segments, fails to pass through the node indicated by the red markers.

Figure 48. Example of Geographical Error Visualization

6. Summary of Functional Testing

The functional tests support verification of the following:

- The Annotated Mobility Graph correctly encodes the Enemy Force units that present risk to its individual nodes
- FireSupportTasks are constructed and executed correctly
- The PartialPlan and resulting BattlePlans are correctly structured and sequenced
- The RiskIntervalSubregion class correctly computes base risk intervals
- The RiskInterval class correctly subdivides RiskInterval objects at required time points
- The RiskIntervalSubregion class correctly stores and updates residual risk values
- The AssaultAction and SupportByFire classes correctly compute residual risk values when applied to a RiskIntervalSet

- For each AvailabilityTask, the PlanFireSupport() function generates a potential FireSupportTask against every RiskInterval associated with the plan, except where those tasks are impossible to execute

These claims are further supported by the more encompassing quantitative and qualitative testing described below.

C. SCENARIOS FOR QUANTITATIVE AND QUALITATIVE TESTING

The quantitative and qualitative tests are both based on a set of combat scenarios. Each scenario has three defining features: a force pairing, a map, and a planning type. A *scenario group* is defined by just the force pairing and map. We describe the anatomy of a scenario in this section. Detailed descriptions of the six scenario groups used for testing are provided in Appendix D.

1. Force Pairings

In the terms of the Conceptual Planning Framework (described in Chapter III), a force pairing is a Friendly Force and an Enemy Force. Before listing the pairings, we first define some unit templates and command templates.

The unit templates are

- Fire Team: four entities with rifles (range 550 m).
- Machine Gun Team: three entities, two with rifles and one with a machine gun (range 1800 m)
- Observation Post: two entities with rifles
- Command Unit: one entity with a rifle

The command²⁶ templates are

- Rifle squad: one command unit (squad leader) with three subordinate fire teams. Since attacking squads operate as a single formation, a Friendly Force rifle squad counts as one formation to the planner. Defending units are considered individually, so an Enemy Force rifle squad counts as four formations to the planner: one single-entity command unit and three fire teams.

²⁶ As defined in Chapter I and III, and in contrast to the traditional usage, we use *command* to mean a collection of units with hierarchical relationships at any echelon.

- Machine gun squad: one command unit (squad leader) with two subordinate machine gun teams. Machine gun squads are employed in a single formation and are only used by the attacker in these tests, so they always count as one formation to the planner.
- Infantry platoon: one command unit (platoon commander) with three subordinate rifle squads. Attacking platoons can move as a single formation but must break into squad formations to assault. Platoon command units are usually placed in protected positions during the assault, so they tend not to generate any risk intervals. Therefore, a Friendly Force platoon counts as three formations to the planner. An Enemy Force platoon counts as 13 formations: a command unit plus 4 units per subordinate rifle squad.
- Infantry platoon (reinforced): one command unit (platoon commander) with three subordinate rifle squads and two subordinate machine gun squads. The machine gun squads operate separately from the rifle squads, so this command counts as five formations to the planner. Only the Friendly Force uses this command template in our tests. Reinforced platoons are only used for the platoon-versus-squad scenarios.
- Weapons platoon: one notional command unit (not tasked in the maneuver plan) with three subordinate machine gun squads. This counts as three formations for the planner, and is only used by the Friendly Force.
- Infantry company (Friendly Force only): one notional command unit (not maneuvered) with three subordinate infantry platoons and three subordinate machine gun squads. This counts as 12 formations to the planner: 3 per platoon and 1 for each machine gun squad.
- Infantry company (Enemy Force only): one command unit (company commander) with three subordinate infantry platoons. This counts as 40 formations to the planner: 1 command unit and 13 formations per platoon.
- Battalion (Friendly Force only): one notional command unit (not maneuvered) with three subordinate infantry companies and one subordinate weapons platoon. This counts as 39 formations to the planner: 12 for each infantry company plus 3 for the weapons platoon.

In addition to the units listed above, every defending Enemy Force includes one observation post, adding one to the number of formations for which the planner must account. Machinegun squads are exclusive to the Friendly Force, and observation posts are exclusive to the Enemy Force.

The force pairings are based on an approximate three-to-one numerical advantage for the attacker. Traditional military wisdom calls for these odds before launching an attack. This is meant to overcome the defender's advantages, which include selection of fighting positions and engagement areas, fortifications, obstacles, target refinement, stable firing platforms, and economy of movement. Since the commands are built using the "rule of three"—meaning that each command unit has three subordinate units—a three-to-one advantage is created by providing the Enemy Force with a command template one echelon below that of the Friendly Force. In the following list, the force pairings are described as " F versus E ($x:y$)," where F is the command template for the Friendly Force, E is the command template for the Enemy Force, and x , y are the respective number of formations (not entities) that the fire support planner must consider. The attacker's 3:1 odds do not translate to the formation counts because the Friendly Force has larger formation sizes than the Enemy Force, and its command units are not considered during tactical planning. Defending command units, on the other hand, will be encountered during assaults and are considered individual formations. We are interested in the number of formations considered by the fire support planner, as opposed to the number of entities in each force, because the planner works at the formation level.

- Infantry Platoon (reinforced) versus rifle squad (5:4)
- Infantry Company versus infantry platoon (12:14)
- Battalion versus infantry company (39:41)

In the first pairing, the infantry platoon is reinforced (note in the above command templates that there are two infantry platoon entries, one of which is "reinforced") to provide some fire support units for the planner to consider. Otherwise, fire support could only be provided by the subordinate rifle squads. Although we could simply select one rifle squad as a fire support unit, making it unavailable to the maneuver planner, the reinforced platoon presents a slightly more complex planning problem to the fire support planner, which is more useful in testing.

The second pairing has only one more machine gun squad than the first, but it is significantly more complex. The planner must consider three times the number of rifle squads in the Friendly Force, and more than three times the number of defending

formations in the Enemy Force. Since each defending formation could potentially affect each attacking formation, we have a roughly polynomial increase in the number of risk intervals to consider. Although terrain can limit this increase, the defenders are intentionally placed with good fields of fire, meaning that terrain occlusions are minimized. When moving from the second to the third force pairing (battalion versus company), the width of the command-level formation exceeds the maximum weapon range of the entities. This reduces the polynomial increase in risk intervals. All fire support units will attempt to generate fire support missions against all risk intervals, but in many of these cases will not be able to get into position early enough to affect them. These discrepancies are not well captured in the assumptions for our theoretical upper bound, a fact that highlights the importance of quantitative testing.

The observation posts are meant to provide an additional challenge for the fire support planner. Normally, observation posts are placed apart from the main defending force to cover dead ground and provide early warning of an attack. Observation posts are usually the first threat to be encountered by an attacker, and their elevated positions tend to generate longer-duration risk intervals than other units. If the planner expends too many resources suppressing a small observation post, critical suppression of the main force, and success of the attack, will suffer.

These three force pairings are meant to test the increase in planning time caused by an increase in the number of formations. This appears as n in the asymptotic analysis. Additionally, it stresses the fire support planner's effectiveness in dealing with an increasingly complex tactical scenario. Since the current planner implementation does not reason above the platoon level, but human fire support planners do, the larger force pairings offer a chance to determine whether the planner requires more hierarchical structure to its reasoning.

2. Maps

Another factor in the asymptotic analysis is v , the number of nodes in the navigation graph. To see how an increase in the size of the Annotated Mobility Graph affects the fire support planner's running time, we use three different maps of different

sizes. To simplify the comparison, we maintain the same lateral scale of elevation postings for each map. In other words, the distance from one elevation posting to an adjacent one, projected onto the xz plane, is a constant number of meters for all maps.

To support the argument that the maps are tactically relevant, we import real world elevation data using the Real World Terrain tool (Infinity Code 2016), which was procured from the Unity Asset Store. Tactical wisdom suggests that units and tactics perform differently on different classes of terrain—desert, rolling hills, mountain, etc.—and clearly a very flat terrain will result in a different nature of risk interval sets than a terrain with many small obstructions, where formations can pass in and out of view many times. Although varying the terrain class would serve as an interesting experiment, this is not the focus of our current testing plan. To mitigate situations where the terrain class confounds results about the terrain size, but still work with some interesting variety of tactical situations, we restrict our focus to maps with all the following characteristics:

- Rolling hills of varying height with elevation changes several times the height of an entity
- Valleys with long areas of relatively low elevation and ridgelines on both sides
- At least one dominating hilltop feature with a maximum elevation greater than other nearby features

These types of maps are often found in the foothills of mountainous regions. There is a large supply of such areas in the U.S., and U.S. elevation data tends to have the greatest quality and resolution. Each map is drawn from a different U.S. state. Figure 49- Figure 51 provide a visualization and size for each map.

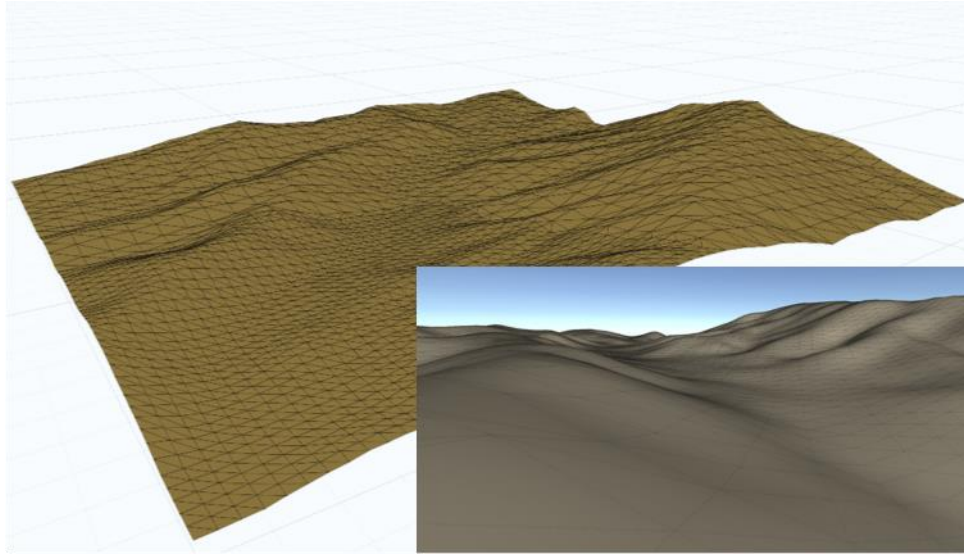


Figure 49. Terrain A. Cayucos Creek, CA: 2 km², 8192 triangles

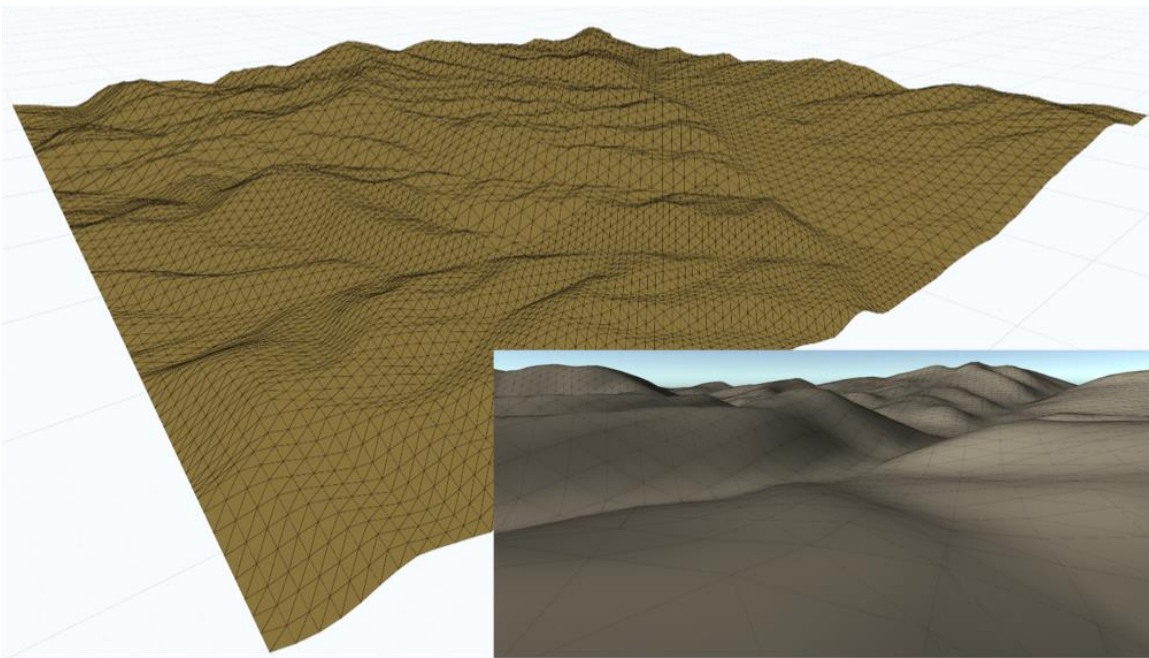
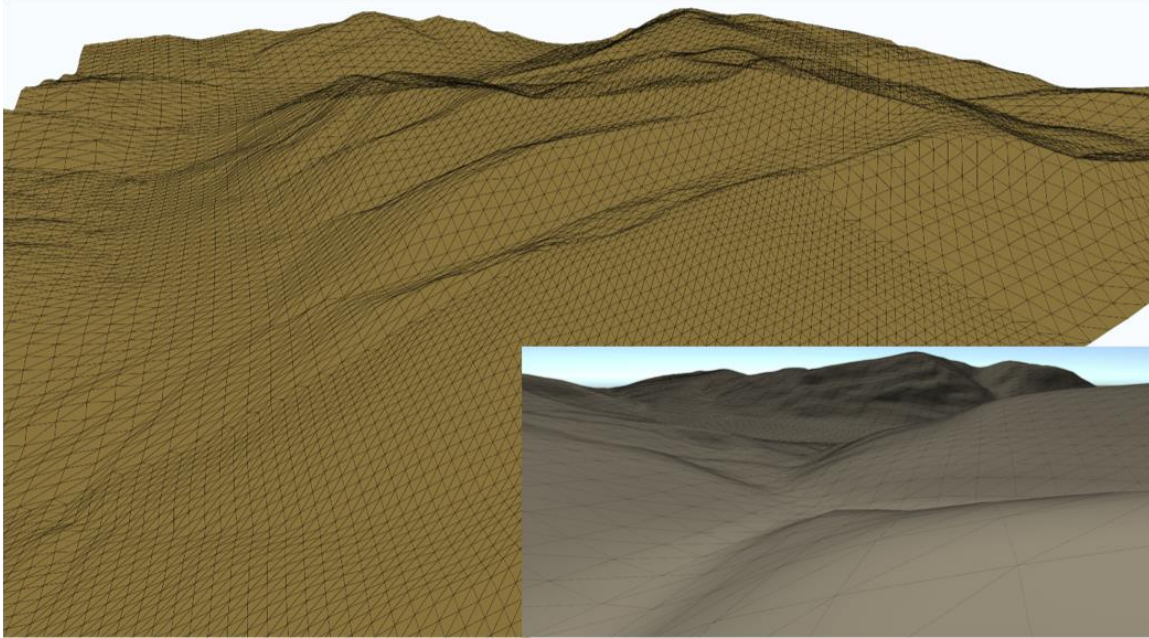


Figure 50. Terrain B. Palo Alto, VA: 4 km², 32,768 triangles



This map has a significant amount of microterrain not apparent at this zoom level.

Figure 51. Terrain C. Buckeye Peak, CO: 8 km², 131,072 triangles

For these experiments, the maps have only one type of terrain surface: bare dirt. We do not import vegetation or water information, and we do not include any man-made structures (no buildings appear on the selected geographical areas). The only obstructions for targeting are the triangles of the terrain skin and the entities themselves.

3. Planning Types

Each scenario group is tested with a few different types of planning. These types only relate to fire support planning. The maneuver plan is the same for each of these types, with the exception of the additional tasks described in the manual planning type. The different planning types are

- *Manual.* In this case, the fire support planner is not invoked. Instead, all fire support units are provided with a single “manual” fire support position as part of the scenario design. Each manual position is selected to allow targeting of multiple Enemy Force units that can threaten the maneuver plan. As part of the maneuver plan, each fire support unit is tasked to follow a tactical path (generated from the same pathfinding module used by the fire support planner) to its manual position. The fire support units do not use FireSupportTask OperationalTasks in this mode; instead, they

use reactive targeting (the same mode used by the defending units) to fire on the closest unit at any given time.

- *Max.* The fire support planner is run until the potential task set is empty. Tasks continue to be added to the plan even if every task score is negative.
- *Best.* The fire support planner stops at the iteration number with the best plan score. Since the fire support planner is deterministic (and the entity starting positions remain fixed), the best iteration number is determined by reviewing the stepwise plan scores from the “max” planning type. It is possible to automate this selection process with a single backtracking point, as described in Chapter IV, but we do so manually for these tests.
- *Best = Max.* In some cases, the final iteration of the Max planning type has the best score. We use this label for those situations.
- *Fastest.* This type is only used with the largest force pairing. The fire support planner is set to stop at 12 iterations, which is significantly less than the “Best” number of iterations.
- *Fast.* This type is only used with the largest force pairing. The fire support planner is set to stop when its total processing time after the next iteration is predicted to exceed a threshold of 120 seconds. This is less than the amount of time needed to achieve the “Best” number of iterations. Since resource-competition among computer processes can vary the amount of time that the planner takes per iteration, this planning type does not always result in the same number of iterations.
- *Close-quarters.* This “planning type” is actually an alteration of fire support unit starting positions, which is discussed in the next section. Close-quarters scenarios are only run with the number of iterations that produces the best plan score. If that number is less than the maximum number of iterations, the planning type is labeled Close-quarters (Best). None of the Friendly Force maneuver units or Enemy Force units are repositioned for Close-quarters scenarios.

Planning types are designed to help answer the following research questions.

- (1) Is the fire support planner implementation able to produce plans with better performance than simple, human-generated plans? We answer this by comparing the manual planning type to all other planning types.
- (2) Is the plan score (the residual risk value) a good predictor of combat results? Within a scenario group, each planning type results in a fixed plan score, which is compared with the combat performance results.

- (3) Does the fire support planner produce better plans when run to exhaustion or when run to the best score? Within each scenario group, we compare the Max and Best plans quantitatively and qualitatively.
- (4) Can we still improve combat results over the simple, human-generated plans if the planner does not have enough time to generate a best or max plan? We compare the Fast and Fastest results to the Manual results.
- (5) Do the initial positions of the fire support units make a significant difference in the quality or performance results of the generated plan? We compare the Close-quarters results to the other results.

For each scenario group, we use the Unity Editor to place units on the map. The Enemy Force entities are positioned around an arbitrarily chosen terrain feature, which we call the *objective area*. We use visual inspection to cover as much dead ground as possible, then verify the coverage with the Annotated Mobility Graph visualization. Each individual entity of the Enemy Force is carefully placed to ensure that its view of the engagement area is not obscured by its immediate surroundings. The Enemy Force is arranged to resemble a western military-style battle position of the appropriate echelon. This includes making use of interlocking fields of fire, which means that more than one unit can fire on each potential approach path. We leave an approximately 90 degree “rear” portion of the objective area uncovered (notionally protected by parts of the Enemy Force not modeled), allowing more firepower to be directed towards the front and flanks. Once the main force is positioned, the observation posts are placed outside the main battle position, on a commanding feature that offers visibility of areas that the main force cannot affect. In real world operations, the personnel on the observation post would fall back to the main position upon the approach of the attacking force. We simply leave them in place to present additional risk to the Friendly Force for as long as possible.

The Friendly Force units are positioned in an *assembly area* outside the maximum range of all Enemy Force weapons. Individual Friendly Force entity placement is not important²⁷ because all entities move to their formation positions as soon as the first tasks begin.

²⁷ In real-world combat operations, the command would be placed in a temporary defensive formation to prepare for spoiling attacks or other enemy action. There is no benefit to modeling this since the Enemy Force is not programmed with any such counter-attack behavior.

The implicit mission of the Friendly Force is to clear the objective area of all Enemy Force units. We assign maneuver tasks to infantry platoons and rifle squads using Route objects. Each Route contains, as transform children, one or more GameObject with specific names expected by the maneuver planner. As described in Chapter V, these are used to perform pathfinding and generate maneuver plans. In general, the Routes are manually designed to resemble a higher-echelon form of maneuver such as a frontal or flanking attack. The tactics of each scenario group are described in Appendix D.

Each unit includes a switch to make it available to the fire support planner by generating availability tasks; we enable this for all attacking squads. Machine gun squads are given no maneuver tasks (except to a single position for the manual planning type), so they are available for the duration of the plan. Rifle squads become available for fire support tasks once their last maneuver task is complete. To ensure that the machine gun squads have enough time to reach useful initial positions, we set a maneuver plan start delay (this setting is found on the Planner object). This forces the maneuver units to begin their first tasks at an arbitrary time greater than 0.

To ensure that similar scenarios (for example, the manual, best, and max scenarios for a particular scenario group) are exactly the same except as described, each scenario group is first developed as a single Unity scene. Once the maneuver plan executes as intended and the fire support planner is verified to be functional, we make as many copies of the scene as needed for the scenario group. Each copy is provided a descriptive name, and only the modified settings are changed. For example, the only difference between a best and max scenario (in the same scenario group) is the value of the Planner's Max Processing Iterations field. The best scenario would have an integer, and the max scenario would have "infinity."

4. Summary of Testing Scenarios

Table 3 provides the complete list of testing scenarios. Map A uses only the smallest force pairing, Map B uses all but the largest force pairing, and Map C uses all three. See Appendix D for the details of each scenario group.

Table 3. Testing Scenarios

Scenario				
Group	Forces	Map	Fire Support Planning Type	
1	Plt vs Squad	A	Manual	
			Best = Max	
			Close-quarters	
2	Plt vs Squad	B	Manual	
			Best = Max	
			Close-quarters	
3	Co vs Plt		Manual	
			Best	
			Max	
			Close-quarters (Best)	
4	Plt vs Squad		Manual	
			Best = Max	
			Close-quarters	
5	Co vs Plt		Manual	
			Best = Max	
			Close-quarters	
6	Bn vs Co		Manual	
			Fastest	
			Fast	
			Best	
			Max	
			Close-quarters (Best)	

D. QUANTITATIVE TESTING

This section is a more thorough analysis of the quantitative results first presented by Harder, Balogh, and Darken (2016). The experiments are nearly identical, but they have been re-run to incorporate minor improvements to the planning system and WXXI.

1. Testing Procedures

We conduct all quantitative tests using an Apple Mac Pro (Late 2013) with 2.7 GHz Intel Xeon E5 processor and 64 GB 1866 MHz DDR3 ECC memory. All tests are launched from Unity’s Editor Playback mode. The planner runs to completion in a single graphics frame, so the timing and graphics performance settings of the game engine have no impact on the scalability test. We have not enabled multithreading, so all computation is limited to a single CPU core.

Once each individual scenario file is able to execute without error, we launch the quantitative test from the `MultipleScenarioRunManager` scene. This invokes each scenario, runs it for a set number of replications, and stores the resulting data in a comma-separated-value text file. We run each scenario for 10 replications in the experimentation mode of `BasicEventProcessor`. This allows each replication to run at the maximum possible computational speed. WXXI uses the discrete event system of this class to process all simulated combat events, so the results are invariant to the speed of execution, the graphical frame rate, and any other resource competition within Unity or the computer’s operating system (this would not have been true if we had used the standard Unity Update or FixedUpdate cycle). Each replication runs to a fixed simulation stop time (in simulation time, not real-time), which is selected to allow the entire plan to execute. If Enemy Force entities remain within targeting distance of Friendly Force entities after this point, the high protection coefficient of the Enemy Force tends to result in an unrealistic amount of attrition to the Friendly Force. Rather than implement complex reactive behavior such as follow-on attacks, hasty defensive positions, or withdrawal, we simply end the replication even if entities are still firing on each other.

The quantitative testing is focused on two issues: scalability and simulated combat performance. Data for both are collected simultaneously during each replication. All statistical computations are done using the JMP 12 application.

a. Scalability

Although we have derived an asymptotic upper bound for the running time of the theoretical fire support planner, the practical scalability of the system is not obvious. The

scalability measurements determine the running time of the fire support planner as a function of the size of the navigation graph, which is equal to the number of triangles in the map, and the number of units from the force pairing. The minimum size of a fire support task is fixed at 60 simulation seconds for all scenarios.

Although the planning system is deterministic, we completely regenerate the plan at the beginning of each replication. This provides measures of variance for the resulting computation time data. We measure computation time simply as elapsed real-time, which is partially influenced by competing operating system activity unrelated to the test. This results in lengthier computation time measurements than a strict profiling approach would generate, but the measurements are closer to the waiting time that would be experienced by a user. The variance from unrelated processing is limited by ensuring that no other heavy computational tasks are scheduled during the quantitative tests, and by the fact that the test is run on just one CPU core (eleven other cores are available to service competing processes).

b. Combat Performance

We expect a fire support plan to improve the success of a maneuver plan. We use two measures to help test this: a traditional attrition measure called the fractional exchange ratio (FER) and a mission-related measure that we call the *mission accomplishment score*.

FER is defined as $\frac{B(0) - B(\tau)}{B(0)} / \frac{A(0) - A(\tau)}{A(0)}$, where $A(t)$ and $B(t)$ represent,

respectively, the size of the Friendly Force and the Enemy Force at simulation time t (Helmbold and Khan 1986). Time 0 represents the start time, and time τ is the simulation stop time. The force size is simply the count of entities not eliminated. We do not score entity templates differently; for example, an entity with a machine gun counts the same as one with a rifle. FER is the ratio of destroyed force fractions.²⁸ Since the

²⁸ None of the replications result in $A(\tau) = A(0)$, so we do not need to deal with the potential zero denominator.

Enemy Force's destroyed fraction is in the numerator, a greater value for FER is a more positive outcome for the Friendly Force.

The mission accomplishment score is defined as $\frac{C(0) - C(\tau)}{C(0)}$, where $C(t)$ is the number of AssaultObjective objects with at least one live Enemy Force entity contained in its area. This measures the number of fire team-sized objectives cleared by the Friendly Force. We do not require Friendly Force units to occupy an objective to count them as cleared. This aligns with the doctrinal definition of the *clear* tactical task (as opposed to *seize*), and it allows more flexibility in the maneuver plan without the need for extra behavior such as distributing remaining forces among objectives. For example, we could use one Friendly Force rifle squad to assault two objectives in sequence, and we do not need to send a fire team back to the first objective afterwards. Since the Enemy Force fights to the last and does not move, the only way to clear an objective is to attrit all of the Enemy Force entities occupying it. In that sense, the mission accomplishment score is really another form of an attrition score. However, it better reflects the accomplishment of the mission because it only counts attrition that is geographically localized. To contrast, it is possible to attrit 75% of the Enemy Force without clearing a single AssaultObjective, and it is possible to achieve 100% mission accomplishment with a very balanced FER (meaning that only a few Friendly Force entities survived, but all objectives were cleared). However, these extreme cases are very unlikely to occur.

The fire support plan is not the only factor that affects FER and the mission accomplishment score. Other significant factors include the nature of the terrain, the force pairing, the Enemy Force's defensive plan, the Friendly Force's maneuver plan, and various numerical settings such as the protection coefficient for dug-in entities. Within each scenario group, all of these factors are held constant, allowing us to measure the effect of each planning type. We also use multi-factor ANOVA and parameter estimation techniques to determine the statistical contribution of different planning types across all scenario groups, after the other effects have been accounted for.

2. Summary Results

The quantitative results are summarized in Table 4. Each row represents the mean results of 10 replications for a single scenario. All time measurements are in seconds of real-time. For measures of variance, see the more detailed results of the following sections. The columns of the table are defined as follows.

- **Preprocessing Time:** the time spent constructing the annotated mobility graph. Since this depends only on the map and the Enemy Force positions, we provide a single average time for each scenario group.
- **Total Time:** the time spent in the `PlanFireSupport()` function, which includes initialization and all iterations of the main planning loop. This does not include the scanning or annotation time listed under Preprocessing because a single annotated mobility graph can be used for multiple invocations of the planner—for example, if the user has changed some of the Friendly Force starting positions but not the Enemy Force positions.
- **Initialization:** the percentage of the Total Time spent from when `PlanFireSupport()` is invoked until the start of the first iteration of the main planning loop. It includes construction of the maneuver plan's `RiskIntervalSet` and the `PotentialFireSupportSet`. The latter involves multiple pathfinding calls, so there is some overlap between the percentages shown for Initialization and Pathfinding.
- **Pathfinding:** the percentage of the Total time spent in the Pathfinding module, including construction of `FireSupportPositionFinder` objects.
- **Mission Accomplishment:** the mean of the mission accomplishment score for the scenario.
- **Fractional Exchange Ratio:** the mean of the FER score for the scenario.

We offer a few summary observations here, which are supported by statistical analysis in the subsequent sections.

- (1) The fire support planner runs in less than 1 second for platoon-versus-squad scenarios, less than 11 seconds for company-versus-platoon scenarios, and jumps up to the scale of 1–6 minutes for the battalion-versus-company scenario.
- (2) Preprocessing time is less than fire support planning time in all but scenario group 4, which has the largest map and smallest force pairing.

- (3) The fraction of planning time required for initialization and for pathfinding varies widely between scenarios.
- (4) The time and combat performance score averages reported here are tight estimates of the true means (the standard error is proportionally small in every case).
- (5) For each scenario group, enabling the fire support planner—that is, comparing the manual planning type against any other—produces a statically significant improvement in both combat performance measures.
- (6) Enabling the fire support planner is a globally significant factor when all other experimental factors are accounted for.
- (7) The highest plan score (risk value) is not always associated with the most positive quantitative results.
- (8) Starting fire support units inside the enemy weapons range (the close-quarters planning type) may cause a small change in plan performance, but there is no clear trend towards improvement or degradation.

Table 4. Summary Quantitative Results

Scenario				Scalability					Combat Performance	
Group	Forces	Map	Fire Support Planning Type	Preprocessing Time	Iterations	Fire Support Planning Time	Initialization	Pathfinding	Mission Accomplishment	Fractional Exchange Ratio
1	Plt vs Squad	A	Manual	0.07					36.7%	0.88
			Best = Max		7	0.24	50%	46%	66.7%	5.13
			Close-quarters		7	0.26	47%	43%	80.0%	6.49
2	Plt vs Squad	B	Manual	0.28					56.7%	1.73
			Best = Max		6	0.52	53%	48%	83.3%	19.50
			Close-quarters		16	0.98	24%	37%	60.0%	1.91
3	Co vs Plt		Manual	3.00					26.7%	0.40
			Best		29	4.39	32%	47%	73.3%	1.87
			Max		35	4.72	30%	48%	65.6%	1.65
		Close-quarters (Best)	25		4.48	28%	47%	73.3%	2.59	
4	Plt vs Squad	Manual	3.90					45.0%	0.53	
		Best = Max		4	0.58	42%	87%	87.5%	8.88	
		Close-quarters		6	0.82	29%	89%	77.5%	2.69	
5	Co vs Plt	Manual	7.91					46.0%	0.89	
		Best = Max		15	10.62	31%	90%	75.0%	3.17	
		Close-quarters		15	10.15	31%	91%	69.0%	2.21	
6	Bn vs Co	Manual	21.18					15.6%	0.38	
		Fastest		12	70.29	47%	54%	35.9%	0.61	
		Fast		40.9	119.02	27%	63%	74.1%	2.06	
		Best		109	187.56	17%	67%	84.1%	2.56	
		Max		238	326.99	10%	61%	87.4%	2.80	
		Close-quarters (Best)		136	179.25	19%	64%	93.3%	4.42	

3. Analysis of Scalability Results

We break down the scalability results into preprocessing and fire support planning categories because these activities are not necessarily performed at the same time. Preprocessing involves the construction of the Annotated Mobility Graph, and fire support planning involves the initialization of potential tasks and all iterations of the greedy best-first planning algorithm.

a. Preprocessing

Table 5 provides a more detailed breakdown of preprocessing times. The two components are defined as

- Scanning Time: the time spent producing the navigation graph from the terrain representation, including the nodes, connections between nodes, inter-node distances, and vertical slope measurements. Scanning only needs to be done once per fixed terrain, but we run it with each replication to measure the variance in scanning time.
- Annotation Time: the time spent measuring node visibility for each Enemy Force entity, measuring node viability as a fire support position against each Enemy Force unit, annotating each node with references to Enemy Force units and entities, and computing node penalties using the annotations.

The histograms of these data (not shown here) are all tightly grouped, which is not surprising due to the low standard error measurements. The scanning times show some right skew, which is to be expected since there is a minimum possible time to scan a map, but in theory no upper bound on the extraneous operating system activity that is also captured in the measurements. The annotation times appear more normally distributed, probably because they are longer and must be spread across more context switches.

Table 5. Preprocessing Scalability Results

Scenario			Scanning Time		Annotation Time	
Group	Forces	Map	μ	std. err	μ	std. err
1	Plt vs Squad	A	0.07	0.0013	0.85	0.0077
2	Plt vs Squad	B	0.28	0.0034	1.06	0.0103
3	Co vs Plt				2.72	0.0189
4	Plt vs Squad	C	1.21	0.0094	2.69	0.0235
5	Co vs Plt				6.69	0.0290
6	Bn vs Co				19.97	0.0749

b. Fire Support Planning

Table 6 shows the time required for fire support planning, fire support initialization, and pathfinding. These are the same mean times reported in the similarly named columns of Table 4, but here we show initialization and pathfinding in seconds rather than a percentage of the fire support planning time. With a running time of less than 1 second, the platoon-versus-squad scenarios are fast enough for fire support plan changes in real-time applications. The company-versus-platoon scenarios take several seconds, so a real-time application would need to conduct planning in a separate thread or spread it across multiple frames (such as with Unity’s coroutine functionality). The battalion-versus-company scenario does not exhibit real-time performance speeds, even when limited to just 12 iterations. We expected this type of nonlinear growth in running time due to the large polynomial terms of the asymptotic analysis, but these data provide a useful benchmark for future improvements.

Table 6. Fire Support Planning Times

Scenario				Iterations	Fire Support Planning Time		Initialization		Pathfinding	
Group	Forces	Map	Fire Support Planning Type		μ	std. err	μ	std. err	μ	std. err
1	Plt vs Squad	A	Manual							
			Best = Max	7	0.24	0.0043	0.119	0.0043	0.109	0.0043
			Close-quarters	7	0.26	0.0040	0.123	0.0040	0.113	0.0040
2	Plt vs Squad	B	Manual							
			Best = Max	6	0.52	0.0089	0.275	0.0089	0.246	0.0089
			Close-quarters	16	0.98	0.0114	0.238	0.0114	0.357	0.0114
3	Co vs Plt	B	Manual							
			Best	29	4.39	0.0465	1.388	0.0465	2.059	0.0465
			Max	35	4.72	0.0531	1.431	0.0531	2.269	0.0531
			Close-quarters (Best)	25	4.48	0.0403	1.242	0.0403	2.104	0.0403
4	Plt vs Squad	C	Manual							
			Best = Max	4	0.58	0.0065	0.245	0.0065	0.503	0.0065
			Close-quarters	6	0.82	0.0231	0.234	0.0231	0.728	0.0231
5	Co vs Plt	C	Manual							
			Best = Max	15	10.62	0.0528	3.315	0.0528	9.613	0.0528
			Close-quarters	15	10.15	0.0512	3.177	0.0512	9.194	0.0512
6	Bn vs Co	C	Manual							
			Fastest	12	70.29	0.1847	33.031	0.1847	38.015	0.1847
			Fast	40.9	119.02	0.2403	32.289	0.2403	74.547	0.2403
			Best	109	187.56	0.6560	31.977	0.6560	126.540	0.6560
			Max	238	326.99	0.7925	31.966	0.7925	200.596	0.7925
			Close-quarters (Best)	136	179.25	0.5522	33.497	0.5522	114.175	0.5522

An important observation here is that scenario group 6 has about 3 times the number of units as scenario group 5, but requires much more than 3 times the planning time. The interactions across command lines are the source of this bloom in complexity: many more Enemy Force units can threaten maneuvering units, which increases the number of risk intervals, and each fire support unit attempts to deal with all of them. In many cases, this means attempting to travel to positions that can target units on the opposite side of the battlefield. Real world fire support planning is more hierarchical than this—a commander would rarely consider support missions across unit lines,²⁹ instead preferring to allow subordinate units to use their organic assets only against risks relevant to them. A closer model of real-world planning—that is, planning a battalion’s fire support as three separate company-level plans—appears to offer an improvement in running time. We cannot expect planning for scenario group 6 to require just 3 times the running time of scenario group 5 because the battalion-level assets (the weapons platoon’s machine gun squads), by both doctrine and practice, can be employed anywhere the battalion needs them. The interlocking fields of fire from the Enemy Force will still cause a nonlinear increase in the number of risk intervals, but the planner would be able to ignore many potential tasks against non-local risk intervals by looking at the command membership of the threatened unit. For real-time applications, it may be useful to consider a Plan Controller that updates a battalion- or company-level plan one subordinate unit at a time.

Real-time considerations aside, the fire support planner is clearly faster than a human in the context of generating initial plans for all scenarios. We did not record the time to set up the single positions of the manual plans, but it certainly took longer than 5 minutes—the longest automated running time we recorded—in every case. Setting up a plan of the size and level of detail produced by the system for scenario group 6 (for the “fast” and longer-running scenarios), using only intuitive planning decisions, could easily take days or weeks for a human. It is difficult to argue for a “fair” test of speed and

²⁹ Support missions like this are not directly modeled in the current system. An implicit cross-attachment or adjacent-unit support mission occurs when a fire support unit is tasked to support the actions of a unit that does not reside in the same local part of the hierarchy—for example, a unit in a different company.

performance between human and computer. For example, the human (by the conventions of today's scenario designers) is allowed to run the actual combat simulation to gather precise data about the quality of the plan, while the computer (according to this algorithm) is limited to its own derived models and risk values. It is perhaps more useful to point out that the human and the automated planning system each have particular strengths. A human, augmented by this kind of tool, would likely be able to produce a better-performing and more realistic fire support plan than an automated planner alone, and to do so faster than a human alone. Although the prototype system does not currently have the functionality to allow a user to modify a computer-generated fire support plan, we have set the foundation for this by inheriting the `PartialPlan` class from Unity's `ScriptableObject`, which has some built-in persistent storage and display capabilities.

4. Analysis of Simulated Combat Performance Results

Table 7 summarizes the quantitative results for simulated combat performance. The means shown are the same as in Table 4, but the mission accomplishment score is given as a fraction (rather than a percentage) to support consideration of standard errors.

Table 7. Simulated Combat Performance Results

Scenario				Mission Accomplishment		Fractional Exchange Ratio	
Group	Forces	Map	Fire Support Planning Type	μ	std. err	μ	std. err
1	Plt vs Squad	A	Manual	0.367	0.092	0.88	0.225
			Best = Max	0.667	0.000	5.13	0.601
			Close-quarters	0.800	0.054	6.49	1.362
2	Plt vs Squad	B	Manual	0.567	0.071	1.73	0.177
			Best = Max	0.833	0.056	19.50	3.744
			Close-quarters	0.600	0.083	1.91	0.419
3	Co vs Plt		Manual	0.267	0.030	0.40	0.047
			Best	0.733	0.038	1.87	0.148
			Max	0.656	0.042	1.65	0.138
			Close-quarters (Best)	0.733	0.050	2.59	0.364
4	Plt vs Squad		Manual	0.450	0.050	0.53	0.087
			Best = Max	0.875	0.042	8.88	1.307
		Close-quarters	0.775	0.058	2.69	0.470	
5	Co vs Plt	Manual	0.460	0.031	0.89	0.102	
		Best = Max	0.750	0.031	3.17	0.326	
		Close-quarters	0.690	0.023	2.21	0.129	
6	Bn vs Co	Manual	0.156	0.020	0.38	0.024	
		Fastest	0.359	0.012	0.61	0.021	
		Fast	0.741	0.023	2.06	0.132	
		Best	0.841	0.037	2.56	0.173	
		Max	0.874	0.021	2.80	0.181	
		Close-quarters (Best)	0.933	0.011	4.42	0.222	

Our intent is to use statistical hypothesis tests to show where certain planning types are significantly different from others, in terms of these measures. Our primary concern is effect screening, not regression, because the scores are highly dependent on several arbitrary model parameters such as the probability of hit and the protection coefficient. We use the typical criteria of $\alpha = 0.05$ for all tests.

a. Mission Accomplishment Score

Figure 52-Figure 54 provide histograms of the numerator (an integer value) of the mission accomplishment score $\frac{C(0)-C(\tau)}{C(0)}$ for each scenario. The mission accomplishment score, as we have defined it, is really just a linear transform of the count of objectives cleared.

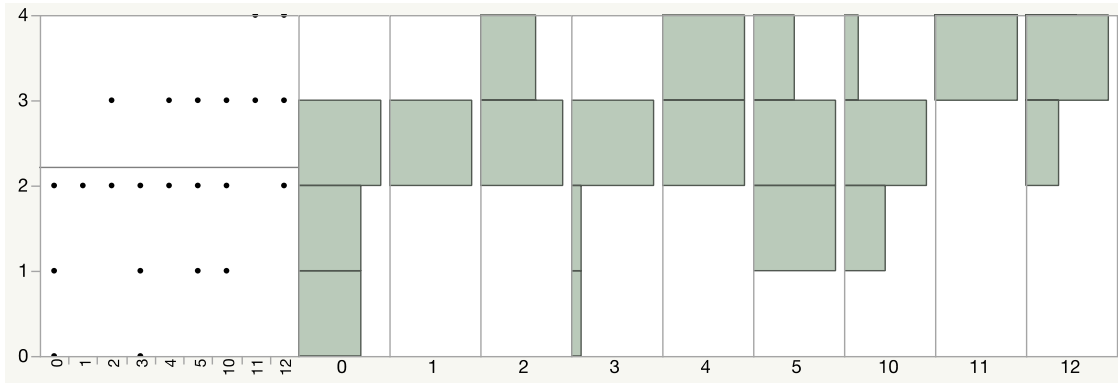


Figure 52. Histograms of Number of Objectives Cleared for Platoon-versus-Squad Scenarios

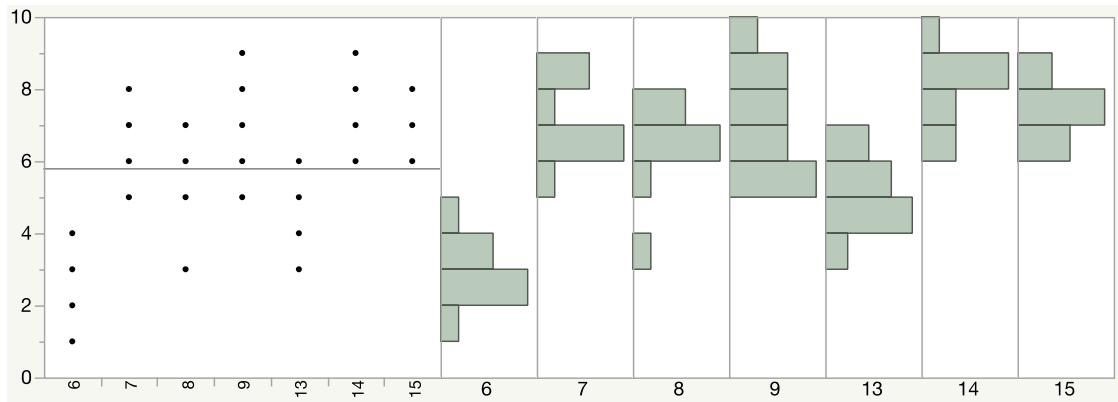


Figure 53. Histograms of Number of Objectives Cleared for Company-versus-Platoon Scenarios

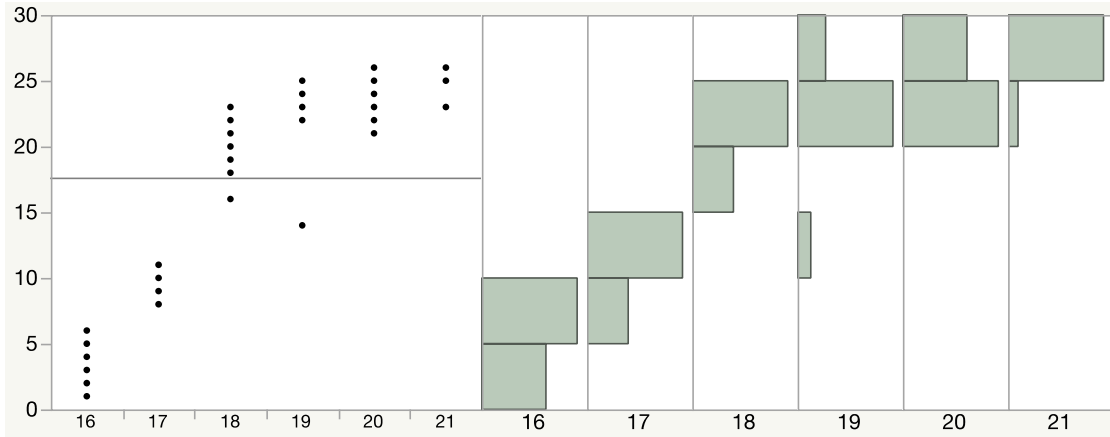


Figure 54. Histograms of Number of Objectives Cleared for Battalion-versus-Company Scenarios

Most of these plots appear fairly symmetric and bell-shaped, but the sample space is discrete, finite, and strictly nonnegative. A normal assumption, for example, would place too much probability mass in artificially infinite tails. In terms of the number of objectives cleared, each replication can be interpreted as a fixed number of Bernoulli trials. If we assume that each trial is independent of the others in the same replication and that all trials have the same, fixed probability of success, then the true distribution is binomial. The bell-shaped histograms of Figure 52–Figure 54 provide further support for this choice. Although the independence assumption is suspect (failure to clear one objective could result in a greater threat to later objectives), we have only 10 replications per scenario, so we do not attempt any more complex discrete probability model than the binomial distribution.

Using the Elastic Net parameter estimation method with a square root grid scale and AICc validation, a binomial regression leads to strong evidence that the automated fire support plans outperform the simple, manual plans in each scenario. The P-value columns in Table 8 are from a Wald χ^2 test where the null hypothesis is that the performance does not differ between the manual and best planning type per scenario. Since different force pairings have different numbers of objectives to clear and therefore a different number of Bernoulli trials, there is no unified binomial regression across all scenarios.

We additionally perform a beta regression on the same data. Although the beta distribution is a continuous model, its adaptability allows a reasonable approximation of the general shapes that we see in the data without much risk of overfitting. Additionally, the beta model is bounded to a finite region ($[0,1]$ in this case for the mission accomplishment score) unlike the normal model. Our beta regression also uses Elastic Net with a square root grid for its parameter search and AICc validation. We reach similar conclusions with this model to those of the binomial regression, but the generalized R^2 values are better for the platoon-versus-squad force pairings. We also perform a multi-factor beta regression across all scenarios, using the force pairing and map as additional effects to account for the variation that they introduce.

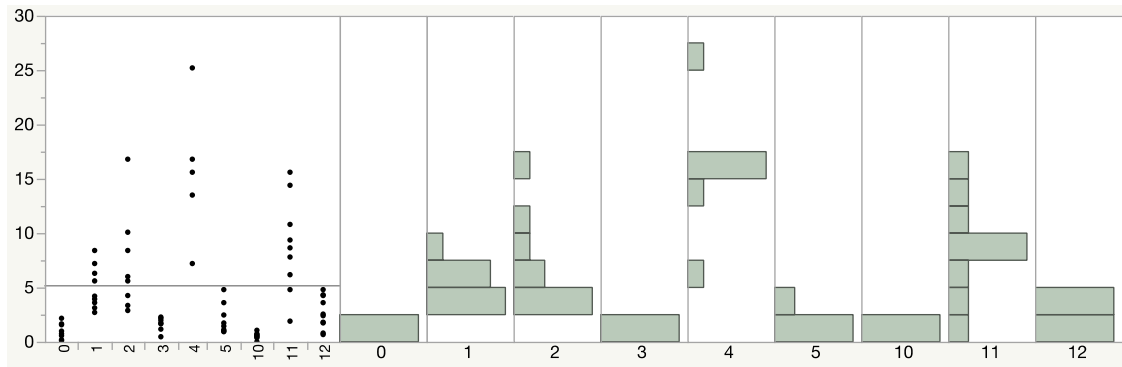
According to the beta model, the close-quarters results do not differ from the best results in a statistically significant way either overall ($P=0.4489$, Gen. $R^2=0.224$) or by scenario, except for scenario group 1 where the close-quarters results are better ($P=0.0007$, Gen. $R^2=0.424$).

Table 8. Mission Accomplishment Score Detailed Results

Scenario				Mission Accomplishment Effect Tests					
				μ	Binomial		Beta		
Group	Forces	Map	Fire Support Planning Type		P-value	Gen. R^2	P-value	Gen. R^2	
1	Plt vs Squad	A	Manual	0.367		0.267		0.424	
			Best = Max	0.667	0.0010		0.0007		
2	Plt vs Squad	B	Manual	0.567		0.261		0.452	
			Best = Max	0.833	0.0042		<0.0001		
3	Co vs Plt		Manual	0.267		0.876		0.796	
			Best	0.733	<0.0001		<0.0001		
4	Plt vs Squad	C	Manual	0.450		0.612		0.699	
			Best = Max	0.875	<0.0001		<0.0001		
5	Co vs Plt		Manual	0.460		0.604		0.697	
			Best = Max	0.750	<0.0001		<0.0001		
6	Bn vs Co		Manual	0.156		1.000		0.898	
			Best	0.841	<0.0001		0.0394		
Overall			Manual	0.378				0.650	
			Best / Max	0.783			<0.0001		

b. Fractional Exchange Ratio

The histograms of FER, which are also grouped by force pairing (Figure 55-Figure 57), are reasonably symmetric and bell-shaped. The accompanying scatter plots, though, call the equal variance assumption into question since variance appears to increase with greater FER scores.



This figure leaves out one extreme data point at 46.8 in scenario 4, which is the Best = Max scenario of group 2.

Figure 55. Histograms of FER for Platoon-versus-Squad Scenarios

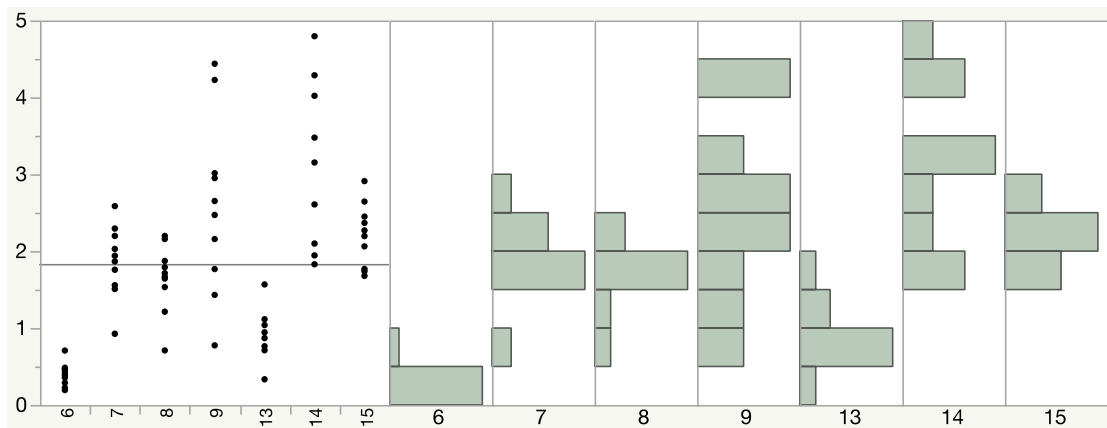


Figure 56. Histograms of FER for Company-versus-Platoon Scenarios

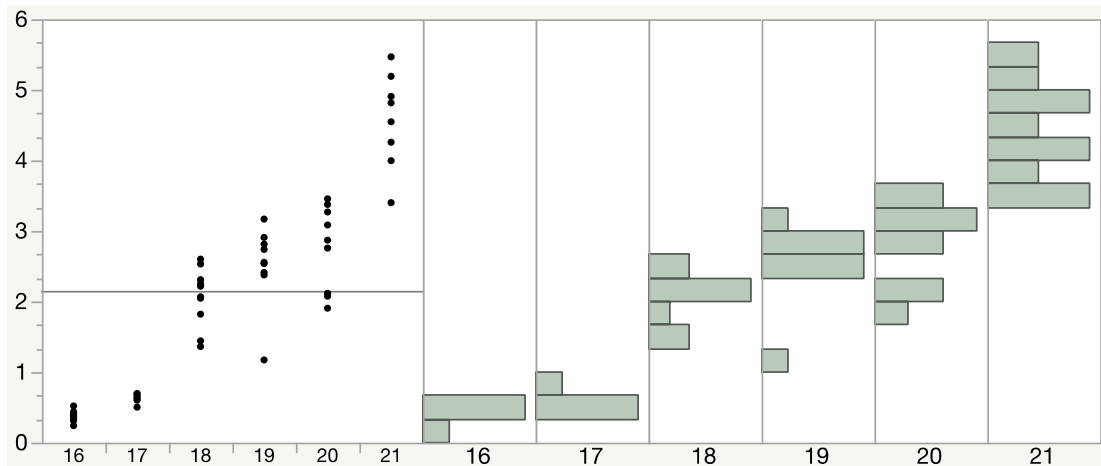
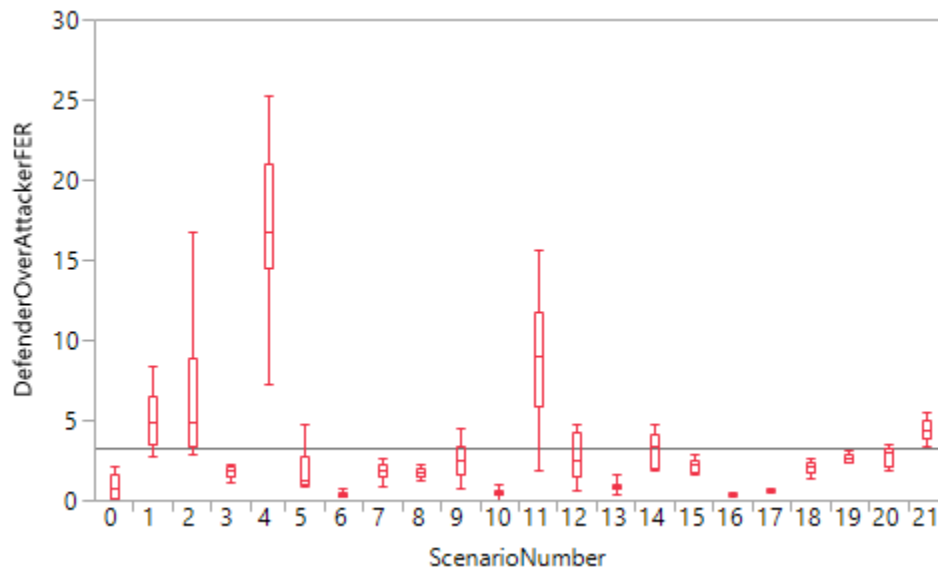


Figure 57. Histograms of FER for Battalion-versus-Company Scenarios

Since it has been previously reported that FER is well modeled by a lognormal distribution (Helmbold and Khan 1986), we compare the original FER data against a logarithmic transform, and find that the latter displays more consistent, though not clearly equal, variance (Figure 58-Figure 59). Since an incorrect equal variance assumption tends to result in false negatives (retention of the null hypothesis, a “safe” error that turns out to be irrelevant), we proceed with ANOVA on the logarithm of FER.



The data point at value 46.8 for scenario 4, again, is not shown here.

Figure 58. Box Plots of FER for All Scenarios

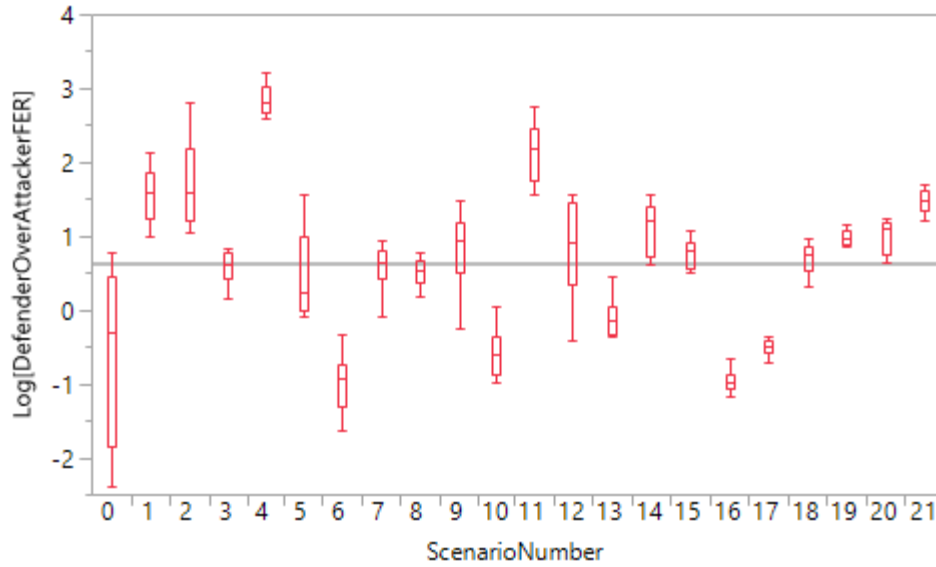


Figure 59. Box Plots of $\ln(\text{FER})$ for All Scenarios

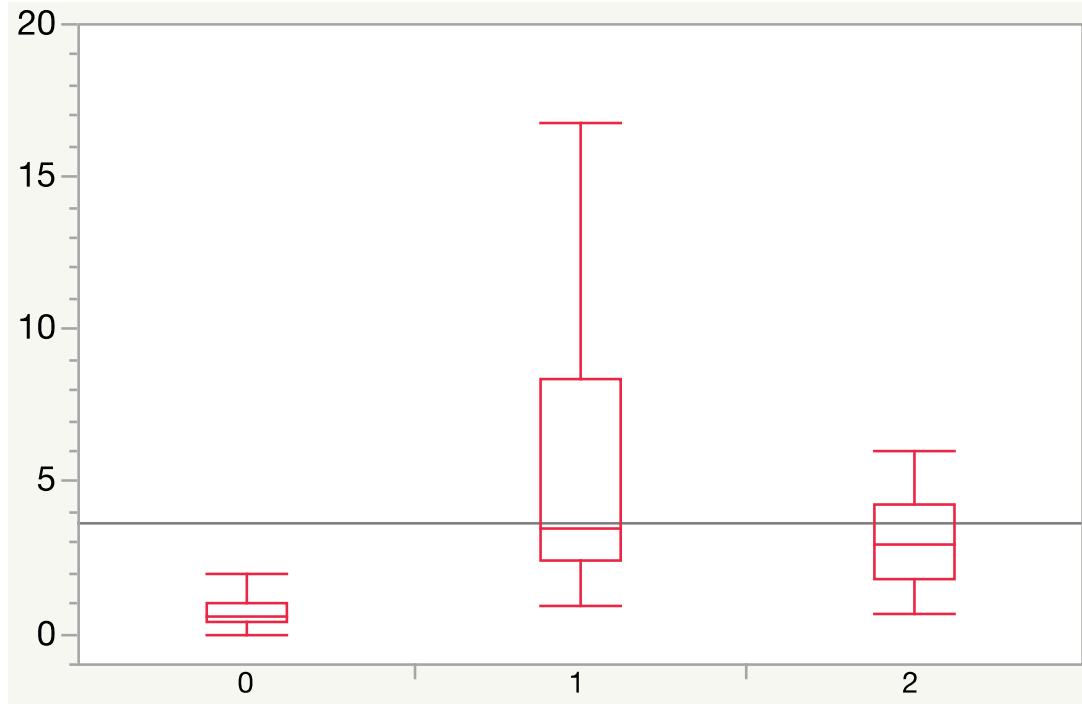
We perform an ANOVA and parameter estimate for the logarithm of FER on each scenario group separately, using only the manual and best planning types. Each of these convincingly shows that the fire support planner results in a better FER score, as shown in the Lognormal column of Table 9. After each of these tests, we add the close-quarters scenario to the model and perform a Tukey HSD test for all pairs of scenarios per group. Unlike the mission accomplishment score, we are able to detect a statistical difference between the close-quarters planning type and most of the manual and best types. Each entry in the Close-quarters column of Table 9 shows whether the close-quarters FER score was better, worse, or statistically tied with the planning type of that row. In all scenario groups except group 2, the close-quarters plan produced a greater FER than the manual plan. Its performance against the best plan is better in one case, worse in two, and tied in the other three—an overall inconclusive result.

Similar to the approach for the mission accomplishment score, we then run a multi-factor ANOVA using the combined results from all scenario groups and adding force pairing and map as additional factors. Not surprisingly, this also shows that the fire support planner dominates simple, manual planning in terms of FER. Overall, the close-quarters results are less impressive than the best planning type, but the difference

between manual and close-quarters is greater than the difference between close-quarters and best. The bottom row of Table 9 shows the results of the multi-factor ANOVA and post-hoc tests. Figure 60 provides a box plot of the FER results by planning type.

Table 9. Statistical Results for FER

Scenario				Mission Accomplishment Effect Tests				
				μ	Lognormal		Close-quarters	
Group	Forces	Map	Fire Support Planning Type		P-value	Adj. R^2	P-value	Trend
1	Plt vs Squad	A	Manual	0.88		0.613	<0.0001	Better
			Best = Max	5.13	<0.0001		0.9135	---
2	Plt vs Squad	B	Manual	1.73		0.861	1.0000	---
			Best = Max	19.50	<0.0001		<0.0001	Worse
3	Co vs Plt		Manual	0.40		0.849	<0.0001	Better
			Best	1.87	<0.0001		0.3639	---
4	Plt vs Squad	C	Manual	0.53		0.878	<0.0001	Better
			Best = Max	8.88	<0.0001		0.0001	Worse
5	Co vs Plt		Manual	0.89		0.754	<0.0001	Better
			Best = Max	3.17	<0.0001		0.0806	---
6	Bn vs Co		Manual	0.38		0.940	<0.0001	Better
			Best	2.56	<0.0001		<0.0001	Better
Overall			Manual	0.80		0.749	<0.0001	Better
			Best / Max	6.64	<0.0001		0.0008	Worse



0: manual planning type, 1: best, 2: close-quarters. Two outliers above the best planning type and three above close-quarters are not shown. This plot is by FER before taking the logarithm.

Figure 60. Box Plots (Less Outliers) of FER by Planning Type

E. QUALITATIVE TESTING

The qualitative tests involve visual analysis of the generated fire support plans with respect to tactics and military intuition. In some cases, the random draws from a single observed replication can produce an uncharacteristically positive or negative outcome. The quantitative tests help the qualitative evaluation by averaging over stochastic combat events; subjective claims are compared against the means and variance of the samples where possible.

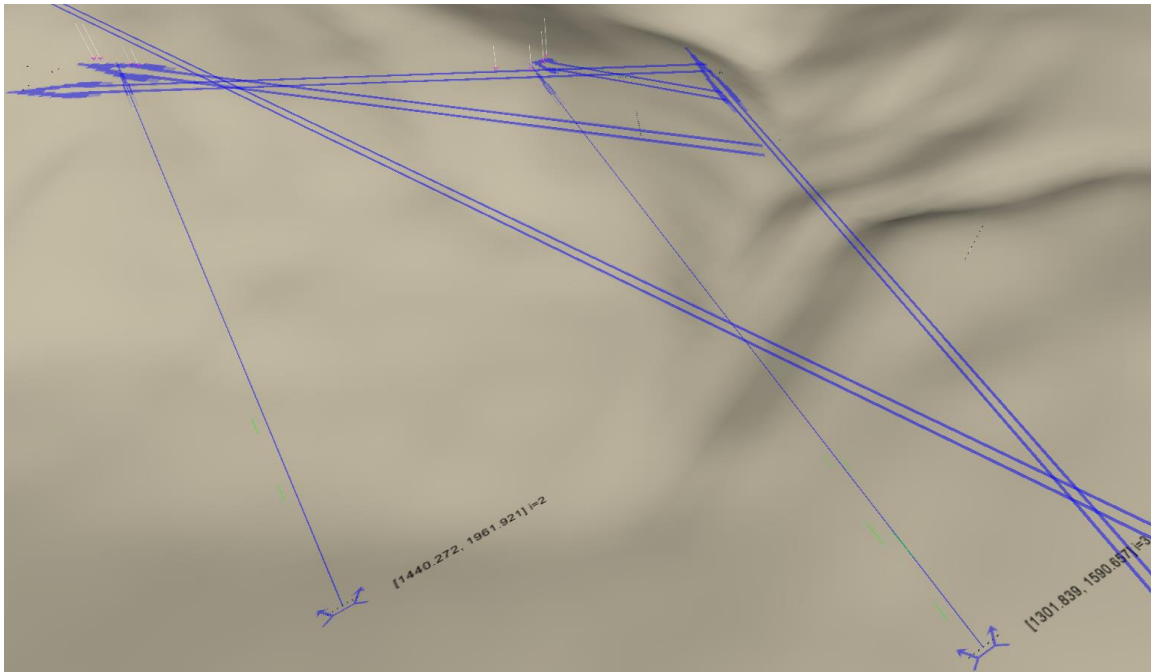
Although we are interested in the scenario and plan features that lead to better or worse combat performance results, the focus of qualitative testing is the appearance of intelligence in the generated plans. By *appearance*, we mean the extent to which plans resemble something that a human might invent, not a claim of any actual thought processes in the planning system. The following research questions further explain the motivations for qualitative testing:

- (1) Is the fire support planner helpful for designing a scripted scenario for an analytical study or training event?
- (2) Would the fire support planner be usable for reactive behavior (planning) in a real-time training scenario?
- (3) What improvements are needed to make the fire support planner's output better resemble human-generated plans?
- (4) Is the fire support planner useful in supplementing the insight of a human planner?

The qualitative results are organized by observed behavioral feature. Despite the relative simplicity of the top-level greedy algorithm, the WXXI CSE, and the minimally featured maps, several interesting tactical decisions arise through automated planning. We discuss both desirable and undesirable features. After describing the features, we present an example of a generated plan and highlight some of those features in the context of a specific tactical situation.

1. Well-Timed Suppression

Suppressing enemy units during times when friendly units are exposed is a critical task for fire support assets (Harder, Balogh, and Darken 2016). This is the fundamental concept on which the algorithm is built. To supplement the statistical results, we observe the plan executions, looking for the key moments when maneuver units are in their most vulnerable positions—usually in the final approach to their objectives. We find that, as long as there are enough fire support units to cover the simultaneous threats, the suppression times align well with the critical maneuver times, as illustrated in Figure 61.



Thin blue lines indicate support by fire tasks, and thick blue arrows indicate general directions of movement.

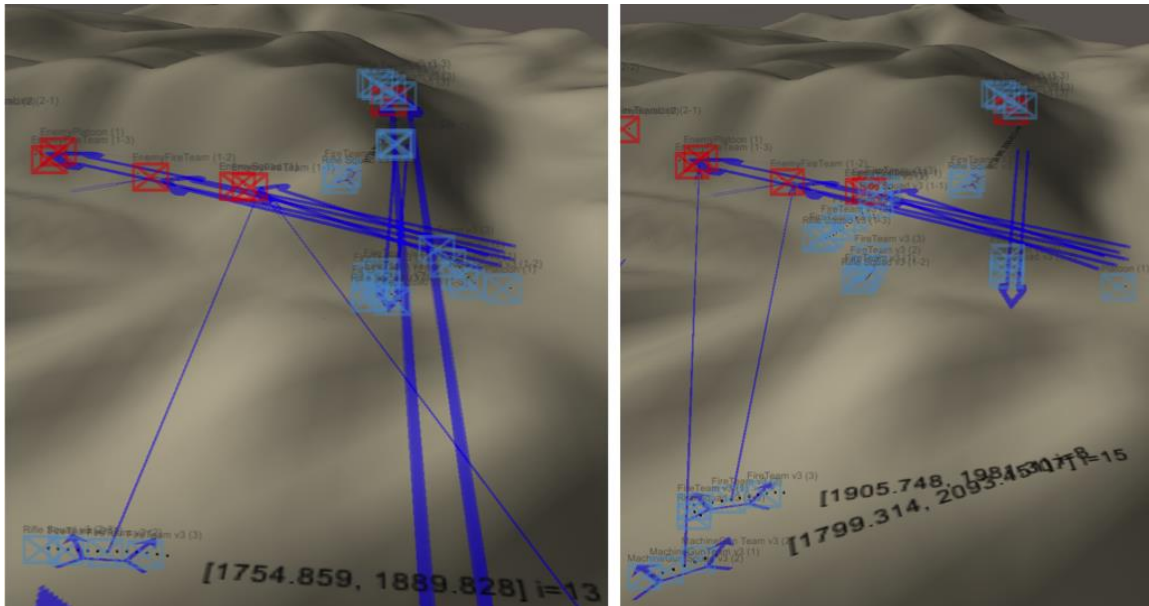
Figure 61. Two Machine Gun Squads Suppress the Objectives as Maneuver Units Begin Their Assault Movements

The advantage of a planning-based approach over a reactive approach to fire support is better coordination and mutual support. This is particularly important when a fire support unit needs to change positions, since it must depart early enough to begin suppression before the threat manifests. Automation relieves the scenario designer of this timing coordination task. Our implementation does not, however, make any changes to the maneuver plan (Harder, Balogh, and Darken 2016).

2. Shifting and Lifting Fires

In a prototypical human-planned assault, the suppressing unit begins by firing on the portion of the Enemy Force that most threatens the advance. When the supported maneuver units get close to their initial objectives, the suppressing unit shifts to other targets to avoid fratricide and to disrupt mutual support from nearby enemy positions. As maneuver units reach their final local objectives, the suppressing unit lifts its fire—that is, stops shooting. Since the risk intervals for route segments close to the objective have

greater risk values (due to the defenders' shorter targeting range), the targets just ahead of the assaulting unit tend to be chosen for suppression. As this plays out multiple times during execution, we can observe the same telltale lifting and shifting behavior that human tacticians tend to use. Figure 62 provides an example.



In the left pane, a machine gun squad suppresses enemy units on an assault objective as maneuver units approach from the right. In the right pane, the maneuver units have closed with that objective and the machine gun squad has shifted to suppress the next objective (and is now joined by another machine gun squad, suppressing the subsequent objective).

Figure 62. Shifting and Lifting Behavior

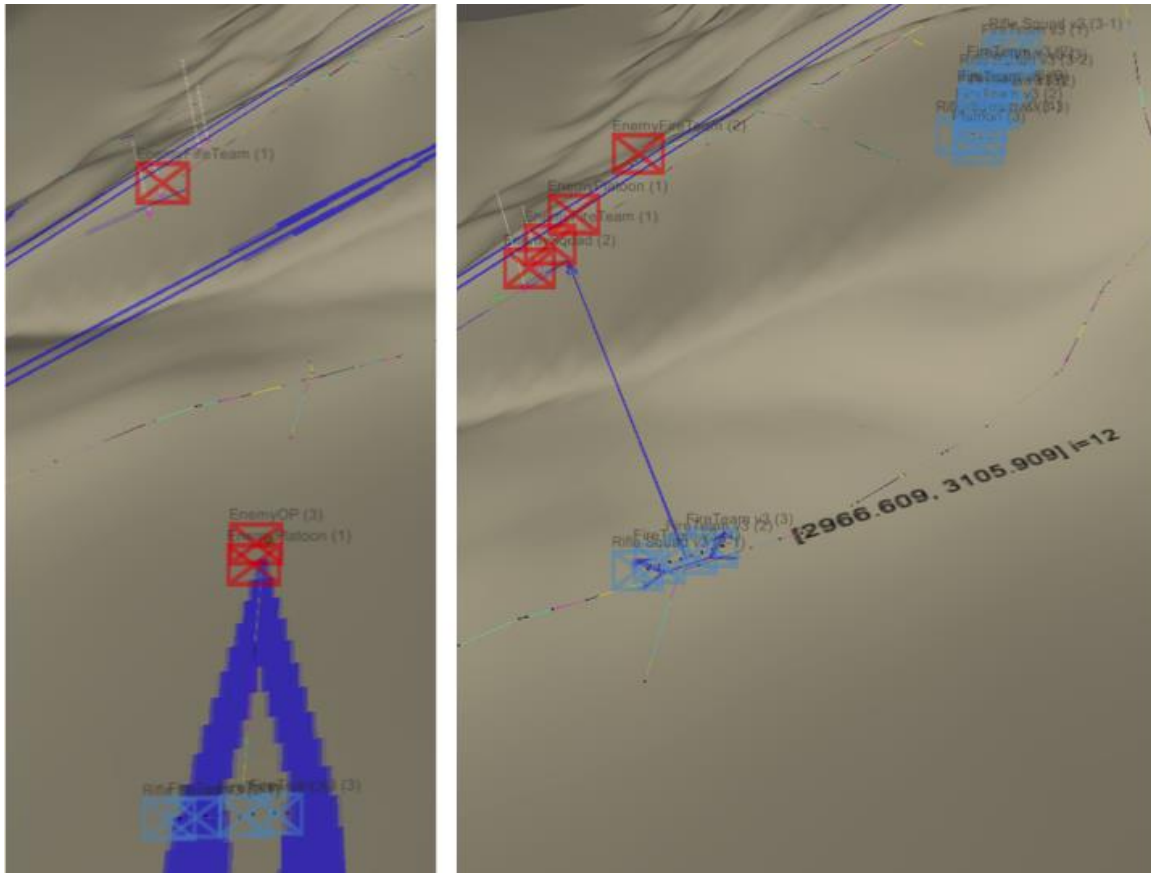
Since we do not directly encode shifting and lifting logic, we view this result as a correct emergent property (Ilachinski [2004] discusses emergence as a form of partial behavioral validation). This result may indicate that our approach to risk mitigation modeling—that is, residual risk values applied by fire support tasks to risk intervals—approximates the thought processes that human fire supporters follow. That is certainly the intent.

Since shifting and lifting is not directly encoded, if some large, nearby unit that is not on the objective presents a numerically greater risk, the planner will target that threat

first. We argue that this is the correct decision, but it is one that a human designer might miss (Harder, Balogh, and Darken 2016).

3. Supporting Follow-on Attacks

Another interesting property emerges when the maneuver plan calls for objectives to be attacked in sequence. In real world operations, it is common for the first assaulting unit to support the second assault once the first objective is secure (Figure 63). The automated fire support planner usually does exactly that (Harder, Balogh, and Darken 2016).



In the left pane, a rifle squad assaults an observation post. In the right pane, the same rifle squad uses the same objective area to provide support by fire to the next assault from the units appearing in the upper right. In this case, the first assaulting unit moved slightly forward from its original objective in order to gain visibility of the targeted unit. (The appearance of additional red unit symbols is an artifact of camera clipping planes.)

Figure 63. A Maneuver Unit Supporting a Follow-on Attack

Since FireSupportPositionFinder only finds the minimum set of firing positions from any given starting point, the first assaulting unit will use its recently seized objective as a firing position to support subsequent assaults if possible.

This behavior is perhaps more directly encoded in the algorithm than some of the other emergent properties are. Favoring firing positions that require no movement is hard-coded in the selection of each fire support task, and maneuver units do not become available for fire support tasking until immediately after their first assaults are finished. However, it is fortunate that our limited heuristic guidance—which keeps the planner simple and relatively easy to debug—often produces the kind of behavior we would expect to see in the real world. The results would appear less realistic if units made long traversals across the battlefield, seeking a position for some best-scoring fire support task.

4. Consolidated Fire Support Positions

A common approach for the employment of organic fire support assets like machine guns and mortars is to use a *fire support position area*, where multiple units can carry out various tasks against multiple targets while under the control of a single leader. Having physical access to the fire support units allows the leader to ensure the proper execution of the fire support plan or to adjust it as needed, even if radio communications fail. Real-world leaders tend to choose position areas with certain features—they should offer good visibility of different parts of the objective area, provide protection from threats and safe approaches for resupply, take advantage of friendly weapons range, and (ideally) not be so obvious that the defenders will have predetermined targeting solutions against them.

Company- and platoon- level fire support position areas seem to emerge in some of the generated fire support plans. Figure 64 and the right-hand pane of Figure 62 provide some visual examples. There is no heuristic guidance in the algorithm for units to group together like this, but this property appears to be influenced by the following features of the scenario:

- The terrain is imported from heterogeneous real world data, so some terrain features (such as the hill occupied by both machine gun squads in Figure 64) offer noticeably better views of the objective area.

- The fire support units are grouped reasonably close together in their starting positions.
- The maneuver plans tend to focus assault units on particular parts of the objective area, so the risk intervals associated with those parts of the plan highlight the subset of Enemy Force units that present the greatest risk at that time. Due to terrain compartmentalization and maximum weapons range, these units are usually grouped closely together, and the positions that can be used to target them also tend to be grouped closely together.

Figure 64. Emergent Company Fire Support Position

So-called *bounding* movement is a technique where two or more units take turns moving and covering. The covering unit takes a position that allows it to support by fire. This is particularly important for fire support units, because moving all fire support assets simultaneously would result in an unprotected maneuver force. Note that bounding for fire support units (sometimes called “leap-frogging”) is slightly different than bounding for maneuver units. Maneuver units tend to cover each other while moving in pairs. Fire support units focus on covering the supported maneuver units, not each other.

Bounding movement for fire support units emerges in the generated plans. The fire support planner usually avoids complete lapses in suppression by moving some units while others are tasked to fire. Bounding movement is likely a result of generating risk intervals for the movement portion of fire support tasks. Once a task including movement is added to the partial plan, potential tasks to cover that movement by fire are generated and considered in subsequent planning steps.

6. Avoiding Threats

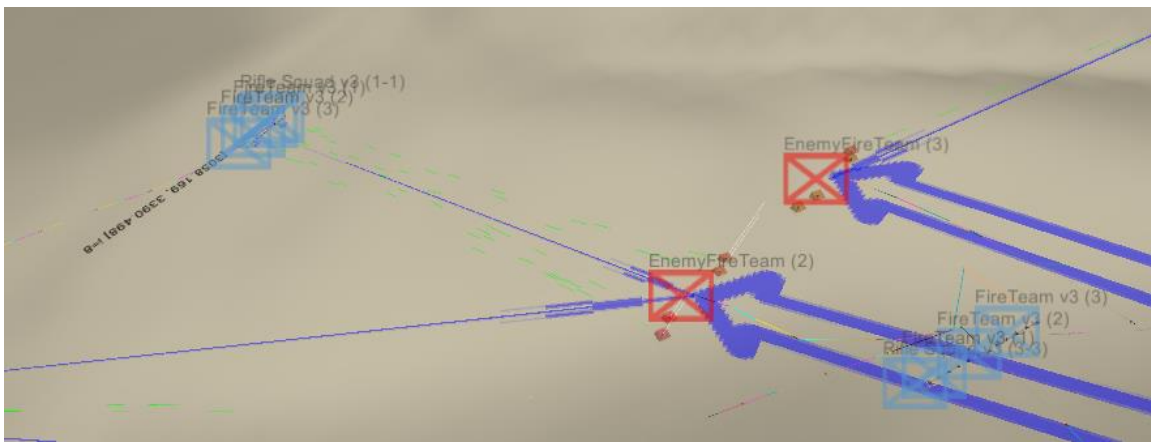
The close-quarters scenarios are not as disastrous as anticipated. This is a benefit of using tactical pathfinding (van der Sterren 2002) during the fire support position search. Since node penalties are weighted much more heavily than distance, the search usually finds a position to target each threat from a relatively safe area. If there is no safe path to a safe position, FireSupportPositionFinder will eventually be forced to choose an unsafe node on the search frontier. Since visibility is somewhat bidirectional (a node threatened by some unit is often selected as a position from which to engage that unit), one of the first few unsafe nodes may be selected as a fire support position. Any subsequently found nodes that could be used to target that same unit are ignored, even if they are safer than the first one. The algorithm is designed in this way for computational speed more than tactics. There are a few different ways that it could be improved, but this issue does not seem to arise outside the rather contrived initial positions of the close-quarters scenarios.

7. Ignoring Low Threats

The plan-space approach to planning and the scoring of risk intervals prevent the planner from getting distracted by relatively low-risk threats, in particular those that the Friendly Force will encounter early in the plan. Some of the defender's observation posts are positioned to test this capability, and we find that they do not cause significant problems. A forward planner, in comparison, might get its resources tied up with a small unit early on and then find a shortage of available firepower available once the main enemy force is encountered.

8. Unsafe Directions of Fire

Since the fire support planner does not account for intervening units when selecting targets, it sometimes produces fire support tasks through, on top of, or towards friendly units (see Figure 65). Real world units, however, consider the surface danger zone (SDZ) of each friendly weapon system to prevent fratricide. Direct fire units, such as the machine gun squads in the prototype, prefer a 90-degree angle between their direction of fire and the direction of movement of supported maneuver units, and the plan should never specify a support by fire interval during which a friendly unit is along or near the direction of fire, even if beyond the target. An exception to this is possible if the terrain and munition trajectory allow friendly units to pass safely under the gun-target line.³⁰



The machine gun team on the left has been tasked to fire directly towards the assault unit on the right. Any trained human tactician would reject this unsafe direction of fire, even if fratricide from missed and ricocheted rounds were ignored in the executable modeled.

Figure 65. Unsafe Direction of Fire

The current version of the fire support planner has no mechanism to reason about the location of friendly forces besides risk intervals, and it only considers a single risk interval each time it creates a potential fire support task. It is, therefore, not surprising that undesirable directions of fire occur from time to time. The abundance of *safe*

³⁰ This tactic is called *overhead fire*.

directions of fire, which are the norm in these experiments, probably arise from the following properties:

- All Friendly Force units are grouped together in their starting positions with an initial direction toward the objective area of, say, 0 degrees
- Most of the maneuver plans call for some form of flank attack with a final assault direction of approximately 90 degrees or 270 degrees
- Fire support units pick the first safe firing positions they can find, so their directions of fire usually do not vary much from 0 degrees

To detect fratricide threats, the task generation step would need to consider expected direction and proximity to all friendly units in the maneuver and fire support plans during the proposed time interval of the task. There is enough information to make these kinds of computations in the partial plan (from the routes). The additional geometric computation would come with a performance cost, but that could be mitigated by reducing the number of potential tasks to consider. Throwing away potential tasks could result in some risk intervals not being dealt with at all, so a deeper search for fire support positions might be necessary, increasing the pathfinding cost. It is not clear whether this would result in an overall improvement or degradation of efficiency.

9. Firing Position Collocation

In some cases, the generated plan tasks two different units with occupying the same exact position at the same time. For just a few small units, this issue is not as significant as the cooperative pathfinding problem (Silver 2005) would suggest because a unit's "location" is actually an approximation of the locations of its entities. We could simply assign all collocated units to the same formation; their entities would then offset themselves dynamically. However, if several units end up collocated, their extreme formation offsets could prevent them from viewing their intended targets or place some of their entities in an unsafe node. Although we do not find any examples of more than two collocated units in these tests, fixing this drawback would be a useful improvement to the planning system. Searching for alternative positions might result in more interesting and effective plans in addition to solving the poor appearance of units walking through each other.

10. Incorrect Cease-Fire Times

One of the challenging aspects of real world fire support is the timing of the shift or lift. Although related to the directions of fire, this is a different problem because even a perfect 90-degree relative direction of fire becomes unsafe when friendly forces approach the target point. We want to suppress threats as long as possible to protect the maneuver force, but we do not want to risk fratricide.

The prototype does not model this very carefully. Each `FireSupportTask` ends when the associated risk interval ends (unless some other `FireSupportTask`, added to the plan during an earlier iteration, already covers the latter part of that time interval). The risk intervals from assaulted units extend until the assaulting unit is scheduled to reach them—more specifically, until the formation leader is scheduled to arrive at the center of the `AssaultObjective` chosen by the scenario designer. This usually means that the fire support units are still suppressing after the assaulting units have crossed their line of fire. Fratricide is not modeled in the current version of WXXI, so this undesirable property actually results in *higher* quantitative scores for the Friendly Force: some of the defenders are still being suppressed while they are being assaulted, at no risk (artificially) to the assaulting entities.

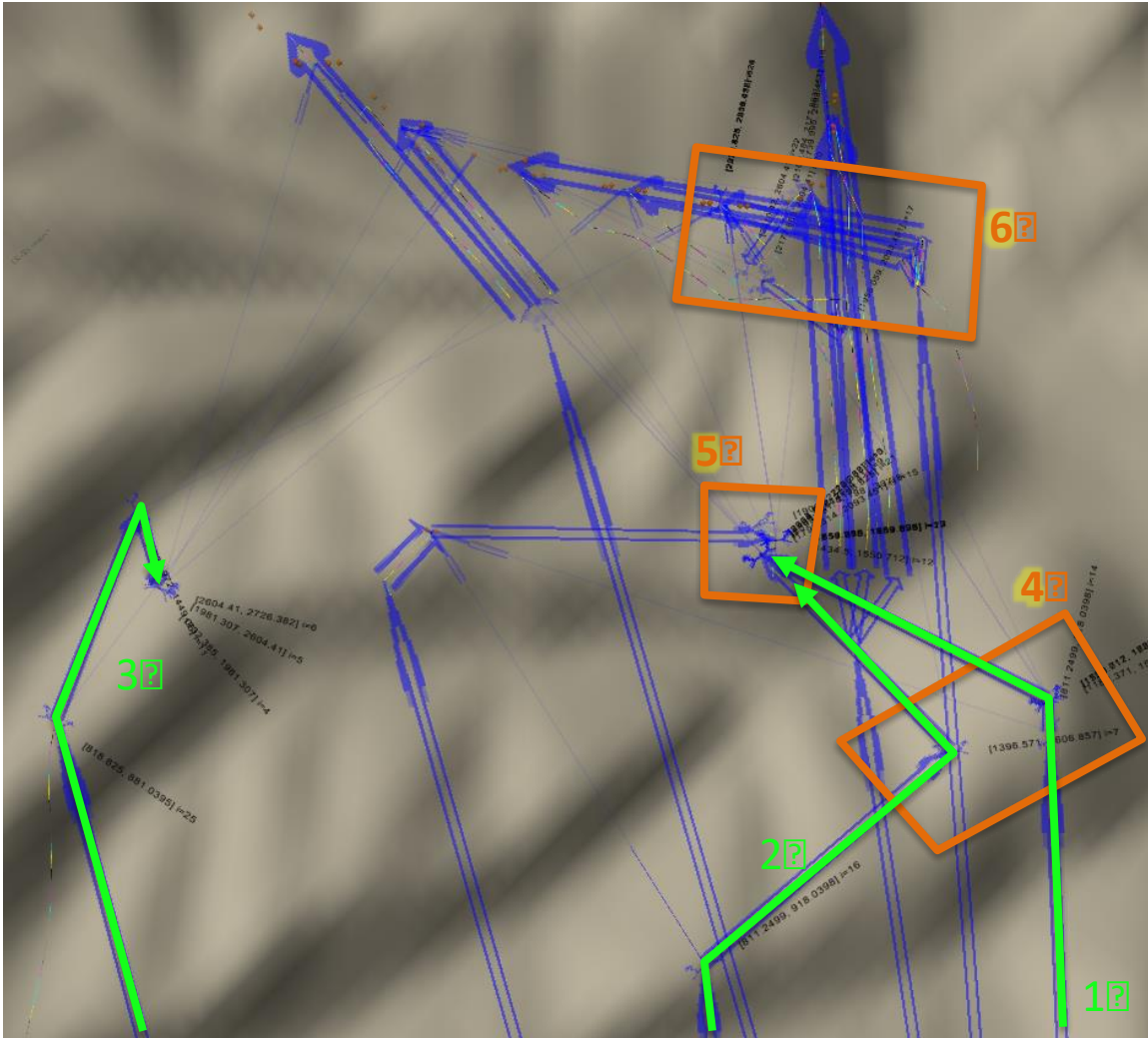
Determining cease-fire times for direct fire weapons, such as the machine guns used in the prototype, depends on the angle between the line of fire and the axis of the assault. 90-degree angles allow the maximum possible firing time, and angles approaching 0 or 180 must be cut off earlier. Precise timing calculations for ceasing direct fire are useless in the real world; instead, visual observation, radio calls, pyrotechnics, and phase lines support event-based decisions. If the assault takes longer than anticipated, the typical answer is to extend the suppression time. As the behavior models in a CSE become more complex and increase the variability of a tactical action's duration towards what we find in the real world, precise cease-fire timing becomes less useful. Although we need an approximation of the ending time for a fire support task to choose a starting time for the next task, we offer that the actual cease-fire decision for direct fire systems is better suited for run-time behavior. If so, then it seems that the simplest approach is to plan the suppression until a slightly later time than the expected

arrival on the objective, similar to what we have done here—then the run-time behavior would only need to decide when to cease fire, not whether to extend suppression.

Indirect fire systems like mortars and artillery require more orchestration and planning to generate effects at the correct time, and their operators typically do not have visibility of the target as direct fire gunners do. Real world indirect fire suppression *does* work with set cease-fire times, but even in this case, the forward observers attached to the maneuver units may need to call for adjustments to the timeline with commands such as *cease loading* or *repeat*. Indirect fire weapons tend to cease fire earlier than direct fire weapons due to their longer time of flight, dispersion pattern, and minimum safe distance. This provides a safety buffer for the assaulting units. Dynamic cessation of fire is therefore an issue for all fire support systems, but simpler for direct fire.

11. Example Plan Output

We present here a description of one fire support plan generated by the system. This example is from scenario group 3, the company-versus-platoon force pairing for map B. We show the best-scoring planning type's results. An overhead view of the combined maneuver and fire support plan for this scenario is provided in Figure 66. We illustrate and describe the tasks for each machine gun squad separately, then point out some of the fire support tasks assigned to rifle squads.



The items labeled 1–3 show the sequence of fire support positions for the machine gun squads. Items 4 and 5 are emergent company fire support positions. Item 6 indicates where several fire support positions are selected for some of the rifle squads to support later assaults.

Figure 66. Example Maneuver and Fire Support Plan Overhead View

The supported maneuver plan is a sequential attack involving the following four stages. The actual plan encoded in the scenario is not broken into explicit stages; these are simply to assist in communicating the ideas to the reader.

- (1) An independent rifle squad assaults the observation post.
- (2) A rifle platoon assaults the eastern defensive positions, moving from south to north.

- (3) A rifle platoon assaults the central defensive positions, starting from the same initial point as the first platoon and moving from east to west.
- (4) A rifle platoon (less one squad) assaults the final, western defensive positions from southeast to northwest.

The most critical moment for this attack occurs at the beginning of stage 2, as the first attacking platoon must approach the objective against the full-strength defenders. The generated fire support plan does an effective job of suppressing the defenders enough to allow this part of the attack to succeed.

Two company-level fire support positions seem to emerge in the plan—the first one (item 4 of Figure 66) supporting stage 2, and the second one (item 5) supporting stages 3 and 4.

Machine gun squad 1 begins on the eastern side of the company formation. Its initial task is to suppress the observation post in support of stage 1. Next, it shifts position slightly to support stages 2 and 3 from the first company fire support position area. Finally, it moves to the second position area to support stage 4. These tasks are highlighted in Figure 67.

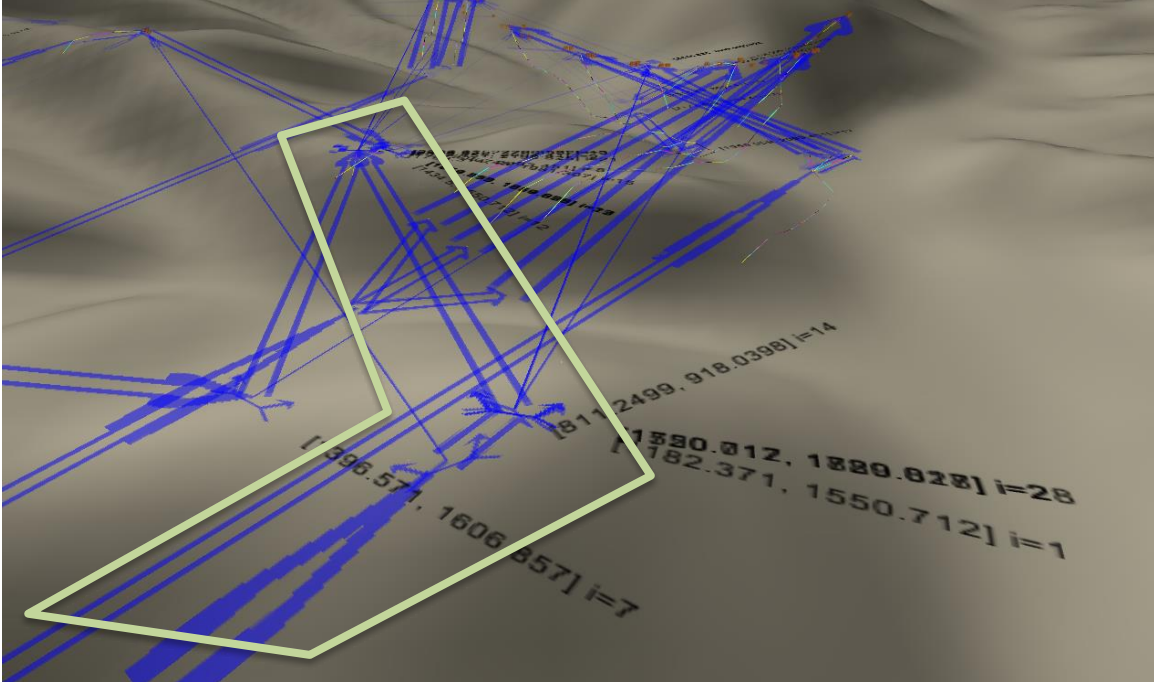


Figure 67. Fire Support Tasks for Machine Gun Squad 1

Machine gun squad 2's task sequence (Figure 68) is similar to that of machine gun squad 1. Since it starts in the center of the company formation, closer to the observation post, there is less distance to cover to support stage 1. Its first movement has a delay to ensure that it arrives at its first position just before its first FireSupportTask begins. The timing of this is more precise than what we would find in the real world (humans would plan to arrive earlier to account for unforeseen events), but as a general rule, just-in-time planning is a reasonable approach when there is risk associated with remaining in a position for a long time.

After supporting the assault of the observation post, machine gun squad 2 moves to the first company position area to support stage 2. It then moves to the second position area to support stages 3 and 4. This is an example of bounding movement because machine gun squad 1 waits until the end of stage 3, after machine gun squad 2 is emplaced, to conduct a similar movement.

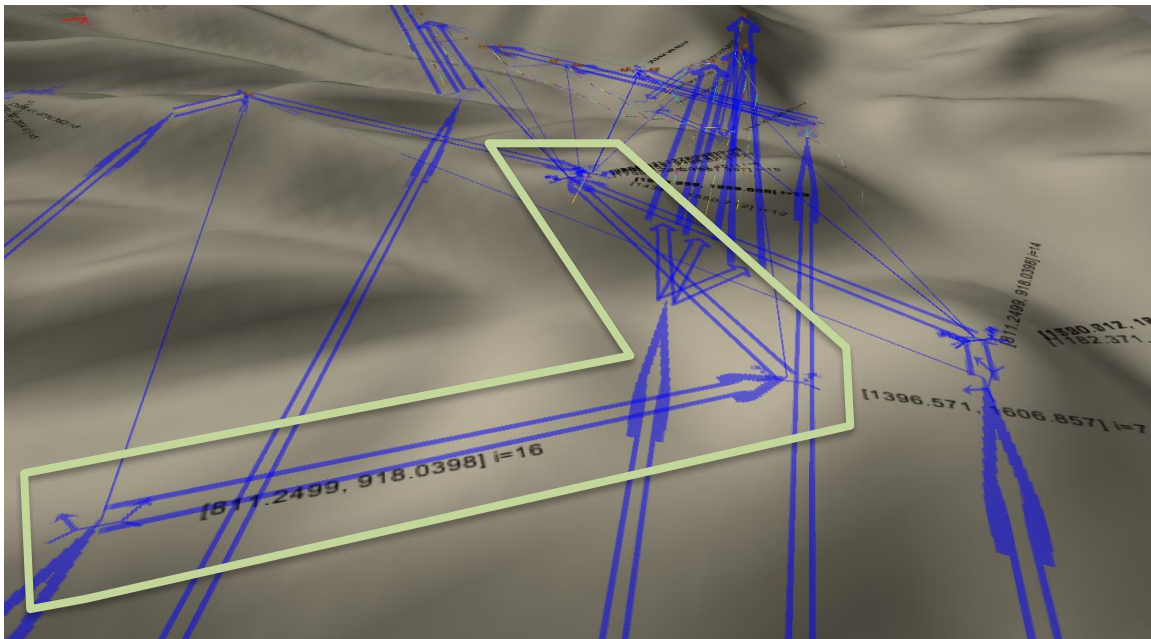


Figure 68. Fire Support Tasks for Machine Gun Squad 2

Machine gun squad 3, which starts on the western side of the company formation, ends up rather isolated—though still effective—in the plan (Figure 69). Although its first fire support position is near the observation post, it does not target it. Instead, it targets an element of the main defensive position that has visibility of the observation post. This helps to protect the independent rifle squad during its assault, since it ends up in the view of the main defensive force as it clears its objective. Machine gun squad 3 then moves forward to support stages 2–4, shifting fire 4 times from right to left as the assaults move from east to west.

The general location of machine gun squad 3 would receive mixed reviews from a human tactician. Its placement allows a near-perfect 90-degree angle with the assault directions of stages 3 and 4. It is also well positioned for stage 4, whose objectives are somewhat difficult to target from the company firing position areas mentioned earlier. However, placing a tactically valuable squad of seven in an isolated position in front of a known enemy position, right next to an enemy observation post (even though in defilade from it), would be a dangerous choice if the defenders were allowed to move. During stage 1, one might argue that the assault on the observation post provides some mutual

support, but the assaulting unit ends up on the second company position area in stage 2, leaving machine gun squad 3 alone in compartmentalized terrain. The residual risk score has no measurement of “closeness” between the tasked units, and of course it does not reason about potential Enemy Force movement. The simplicity of the planning model seems to fall short of good tactics here.

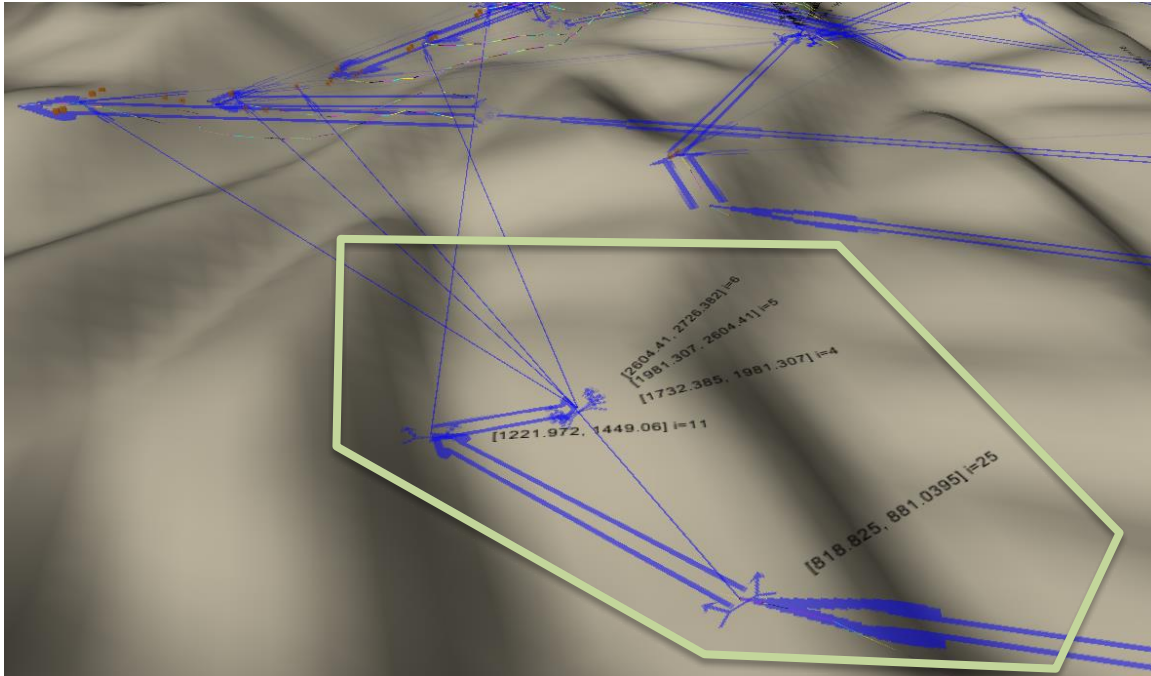
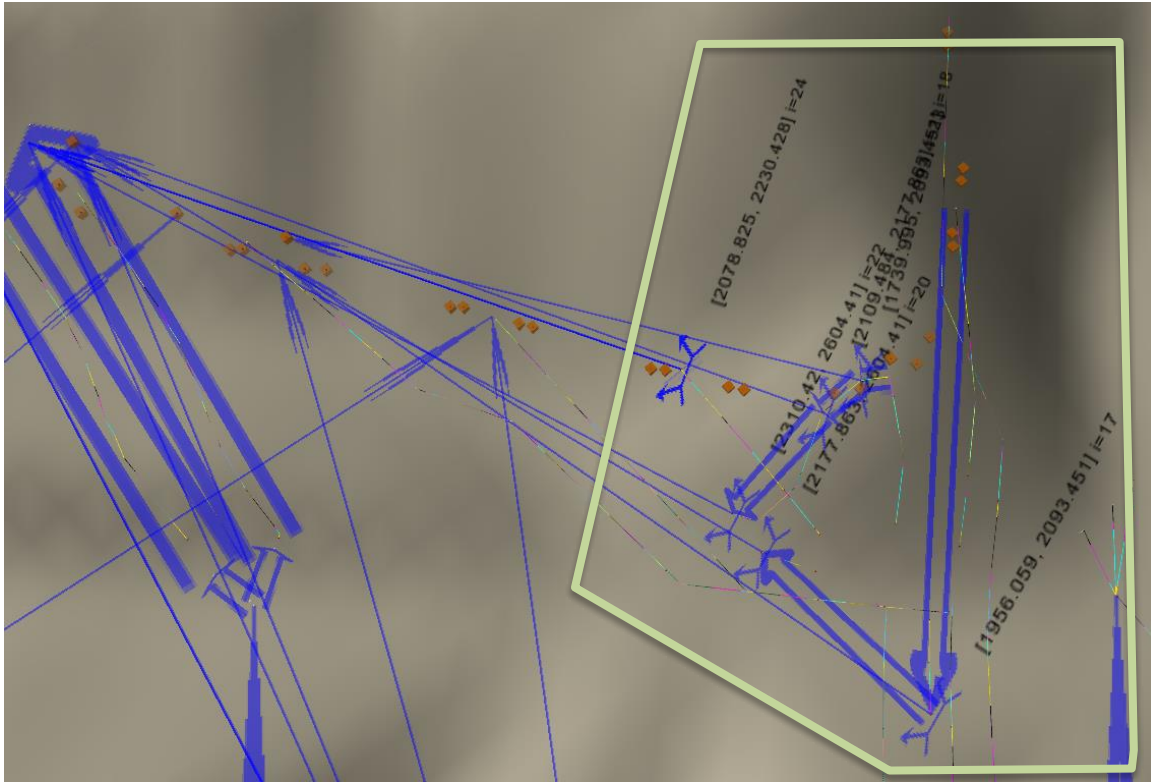


Figure 69. Fire Support Tasks for Machine Gun Squad 3

Some of the fire support tasks generated for rifle squads, after they have completed their assault tasks, are shown in Figure 70. Some of these turn out, as we would expect, to use the position just assaulted as a support by fire position for the next assault. In other cases, such as the transition from stage 2 to stage 3, the assaulted position is obscured from the next objective, so the planner has to generate movement to a new position. The southernmost positions in Figure 70 fall into this category. Unfortunately, these positions are at the base of the occupied hill. We would prefer to see positions that take advantage of the high ground just seized during stage 2, but there is currently no valuation of relative terrain height in the residual risk values.



Some of the maneuver and fire support task symbols have been removed from this image to enhance readability.

Figure 70. Fire Support Tasks for Rifle Squads

The independent rifle squad is the first to finish its assault (in stage 1), so it has more time available for providing fire support than the other rifle squads. As Figure 71 shows, the generated plan sends this squad over to the second company fire support position area, where it shifts fire from right to left ahead of the assaults.

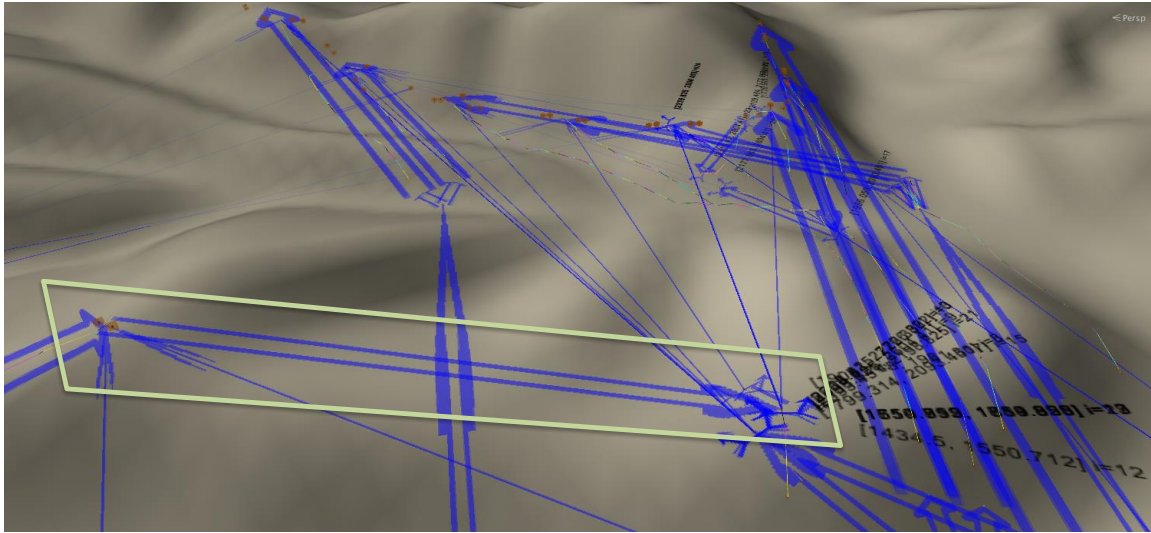


Figure 71. Fire Support Tasks for the Independent Rifle Squad

12. Summary of Qualitative Results

Overall, the generated fire support plans look remarkably coherent, considering the minimal organization enforced by the algorithm. Recall that all of the AvailabilityTasks for squad-sized units are treated as a flat pool of resources by the fire support planner, regardless of their actual hierarchical relationships. To some extent, the coherency and effectiveness of the fire support plans result from the best-first design of the fire support planning algorithm. The first task chosen for each unit is always the one that allows it to provide the best support to the plan, in terms of the PartialPlan's residual risk value. All subsequently added tasks are then constrained by the location and time of the first task. Since traversing all over the battlefield would result in little firing time and minimal improvement (or even degradation) to the residual risk value, the tasks chosen tend to be those with firing positions nearby those already in the plan. Of course, we have also enforced the selection of tasks with no movement at all where possible.

We described emergent features in generated plans resembling techniques that human fire support planners often use, such as shifting and lifting fires, supporting follow-on attacks, consolidated fire support position areas, and bounding movement. The algorithm does well in avoiding danger for the fire support units, even when we craft intentionally difficult situations (the close-quarters scenarios). Some areas for

improvement include unsafe directions of fire, undesirable collocation of units, and the occasional isolation of small units that would hinder their mutual support against potential enemy maneuver.

F. SUMMARY OF TESTING RESULTS

Our three-phased testing shows that the prototype automated battle planning system is effective, adaptive, and bears some resemblance to plans that human tacticians would generate. Although we have not conducted a blind study with multiple human subjects, we have convincing statistical evidence that using the residual risk score is an effective way to improve a battle plan in terms of measures of effectiveness. Our qualitative analysis shows that generated fire support plans are already quite coherent, and it reveals several areas where improvements could be achieved.

VII. CONCLUSION

A. SUMMARY OF APPROACH

We approached this effort with the intention of improving the automation of units in ground combat simulations. After reviewing military tactical doctrine, it became clear that the appropriate focus for this effort is on tactical science rather than tactical art. Exploration of current military simulation systems reveals that battle planning is currently not well automated, and that automated fire support planning in particular has not been attempted despite its importance for real world operational considerations. Automated battle planning systems that exist in the literature, mostly in the gaming domain, tend to advocate a monolithic algorithm, but we hypothesized that planning for each of the various combat functions (maneuver, fires, logistics, etc.) is best supported by a tailored algorithm and unique heuristics.

To provide a context for the addition of a fire support planner—or any other functional planner—to an existing battle planning system, we developed a Conceptual Planning Framework. This organizational scheme offers a separation of planning components and lays out the necessary interfaces and dependencies between them. In particular, each of the tailored planning algorithms (proposed in theory) is placed in a loosely coupled Mission Planner or Enhancement Planner component, and an additional Planning Controller component is conceived to manage the passing of control and information between them. We proposed additional framework components to assist in the separation of duties between system developers and scenario designers, and to improve the configuration management of a planning system that, if implemented in moderate to large scale, could become quite complex.

We then developed a conceptual model of a fire support planner, with the notion that it could work as an Enhancement Planner component subsequent to a Mission Planner component. We modeled the fire support planning problem using a numerical definition of tactical risk, and the reduction of that risk, to a supported maneuver plan. To deal with the large search space of this problem, we developed a greedy best-first fire

support planning algorithm to balance between expected combat effectiveness and computational cost.

To evaluate the fire support planner and begin exploring the utility of the Conceptual Planning Framework, we implemented a partially automated battle planning system prototype. To provide a research-friendly planner development environment, we created the Wombat XXI combat simulator, loosely modeled after the COMBAT XXI system used for analytical studies by the U.S. Army, Marine Corps, and other groups. We created a maneuver planning tool to simulate an automated maneuver planning component through manual data entry (with some automated pathfinding), and provided the output partial plan of this component as a hierarchical task network (HTN) to the prototype fire support planner. This allowed the fire support planner to add fire support tasks to the plan, improving its quality as measured by expected losses.

After performing a variety of functional tests, we evaluated the implementation using combinations of different terrain maps, force pairings, fire support unit starting positions, and limits on the number of planning iterations. We also created simple, manually generated fire support plans for comparison against the plans generated by the automated fire support planner. We ran a series of experiments on different design points, collecting data on running time and combat performance. We performed a statistical analysis on the quantitative data, followed by a qualitative analysis of the behavior resulting from automated fire support plans.

B. DISCUSSION OF RESULTS

We now present conclusions resulting from this research effort.

1. The Case for the Conceptual Planning Framework

Not all battle planning functions are naturally handled by a single planning algorithm. Let us compare the greedy, best-first fire support planner to the maneuver planner of PlannedAssault (van der Sterren 2013). The “manual” maneuver planner of the prototype is built to mimic the internal HTN data structure of a system like PlannedAssault, and the fire support planner takes an HTN maneuver plan as input. But

these two systems use different plan cost functions, and the fire support planner does not adhere to the assumptions necessary for PlannedAssault's A* search. A simple solution for this incompatibility—the one we use in the prototype—is to run them in sequence. There is not much need for a Conceptual Planning Framework for such a simple design, but the prototype is only the first step.

We may wish to integrate fire support planning at a lower level of command and use its results to help score a candidate maneuver plan. Alternatively, we may add other functional components, like a sustainment or communication planner. These may also prove useful early in the planning process: if a planning branch is clearly unsupportable (by logistics, by communications, etc.), then we should avoid wasting computational resources on that branch. Pulling together multiple planning components is nontrivial once we go beyond the two-in-sequence example. One approach is to try melding all components together into a single algorithm and cost function. We may yet find a way to force fire support planning into an A* search compatible with PlannedAssault (see Future Work, below), but what if we then wish to integrate it into some other, different planning system? What are the chances we can force all functional components into that construct? Such a monolithic approach is tightly coupled.

The Conceptual Planning Framework offers a way to design each component with its most natural, manageable, or locally efficient solution. The only required interface is the partial plan format. There are several possible drawbacks to this federation, depending on the implementation details. From the perspective of the integrated whole, we may see some loss of efficiency (even if the components run efficiently in isolation) or reduction in plan quality due to the limited interaction between components. This is a good trade if the alternative, monolithic system's possible efficiency or quality gains are never realized due to unmanageable complexity.

2. The Fire Support Planner and Its Components

The fire support planner is the most immediately applicable result of this work. We have explained all of the necessary data structures, algorithms, and several useful efficiency-related techniques to add a fire support planner to a combat simulation

environment. The conceptual fire support planner is a combination of several techniques, including tactical pathfinding, Lanchester-like equations, and plan-space search. We briefly review here some of the important insights necessary to make it work.

The risk interval concept is central to the fire support planner. The concept is an extension of tactical pathfinding (van der Sterren 2002), whose basic approach is to increase the cost of threatened nodes in a pathfinding search. We extend this approach first by using Lanchester-like equations to estimate the expected losses, measured in number of unit members—that is, the base risk value. Time points and durations in the risk intervals are naturally represented as real numbers. The risk value calculation is dependent on distance from the threats, since closer targets are easier to hit. We can make risk values precise by taking the integral of the killing rate over the expected time for the unit to traverse the risk interval. The second extension from tactical pathfinding is to *reduce* the risk value of each risk interval based on the suppression effects of fire support tasks, which is represented by the killing rate transformations of risk subregions. This reduction simulates the reasoning of human tactical planners, who deal with the often-unavoidable risk of crossing dangerous ground by scheduling suppression against known threats. The sum of residual risk values for a plan give us a numerical score that incorporates both the inherent risk of movement and the mitigation of that risk due to fire support. By adding risk intervals for the movement and firing positions of the fire support units themselves as we add their tasks to the plan, we can judge whether the benefit of a task outweighs its additional risk, and we can naturally represent the benefits of mutual support between fire support units.

We find that risk intervals are best represented with a four-tiered data structure. At the lowest tier, we have risk segments, each of which corresponds to a friendly unit's traversal of a line segment (or perhaps a simple curve) on a threatened portion of one of its planned routes. Each risk segment matches a single pair of units (friendly and threat). At the next tier, we have risk subregions, which are sequences of connected risk segments for a friendly and threat unit pair. By definition, the risk-reducing effects (from the current plan) are uniform for the duration of each risk subregion. Risk intervals, in turn, are connected sequences of risk subregions for the same friendly and threat unit pair. At

the top tier, a risk set is simply a set of risk intervals for any number of different friendly and threat unit pairs. This data structure appears to be a useful tool for tactical problem solving in general, above and beyond fire support planning.

We find that the incremental addition of fire support tasks to a partial plan can, at each step, increase or decrease its risk value (plan cost). If planning goes well, we find a mostly—but not monotonically—decreasing cost. This property is not compatible with a uniform-cost search or A* search, but a greedy strategy is quite effective. With the idea that we should deal with the greatest risks first, we eschew forwards or backwards planning and take a best-first plan-space approach: at each step, we choose the potential fire support task that results in the smallest residual risk value. We add to this a special heuristic: favoring tasks that allow units to remain in the same position. Since units must be located on the terrain surface and have maximum movement speeds, the best-first approach requires reasoning about routes at each step to ensure that all potential tasks are physically achievable, and that irrelevant routes are replaced with the correct ones. This route replacement step is essentially a domain-specific strategy for handling threats to causal links in plan-space planning (Ghallab, Nau, and Traverso 2004, 87–100). The greedy best-first search algorithm is a simplified version of the general chronicle planner for continuous time (Ghallab, Nau, and Traverso 2004, 326–343).

To represent the time intervals still available for tasking, and the locations from which units must travel, we use availability tasks. Each planning iteration amounts to the decomposition of a single availability task into a fire support task and two shorter availability tasks (one before and one after). This decomposition is equivalent to the application of a method in HTN planning. A simple but useful insight is that availability tasks should not be assigned risk intervals, because this could result in useless fire support tasks. By the end of planning, fire support units rarely end up where their availability tasks had them provisionally located.

To create a set of potential fire support tasks for planning branches, we first employ a specialized tactical pathfinding algorithm to generate a minimal set of potential fire support positions for each fire support unit, such that every unit has a position to target every threat, if possible. We then attempt to create a task for each fire support unit

against every risk interval in the partial plan (for its entire duration), including movement to the relevant fire support position along the path already found. If movement time or preexisting tasks prevent the fire support task from covering the entire risk interval's duration, then a task is created for the maximal subinterval unless this falls below a minimum useful duration. We consider this the simplest non-trivial approach because all resources are offered with their greatest available duration against all possible risk intervals using the closest (according to the risk-based tactical pathfinding cost function) set of firing positions, with no consideration of command or support relationships. Other approaches to generating potential fire support tasks are possible, and might improve combat performance, doctrinal realism, or computational efficiency.

Our analysis of the greedy, best-first fire support planning algorithm provides a polynomial upper bound on its running time with respect to the number of units, size of the Annotated Mobility Graph, and maximum number of tasks per unit. This is a higher-order complexity than one might expect from a greedy algorithm, but this comes from performing pathfinding in each iteration and the combinatorics of many fire support units against many defending units, which threaten many friendly units at different times. The minimum allowed task duration prevents unbounded growth in the size of the potential fire support task set. Without this limit, availability tasks could be infinitely decomposed into smaller and smaller pieces, growing the number of potential fire support tasks at each step. Instead, once an availability task is too short to produce a minimal-duration fire support task, it can never be decomposed again.

3. Implementation

Our complexity analysis uses several maximizing assumptions. The actual running time of an implementation depends on factors such as the shape of terrain features, defending unit positioning, and computational hardware. Furthermore, it is difficult to argue for the “correctness” of the greedy algorithm—that is, to assess the combat performance of its output plans—since we admit non-optimal choices and the general solution space is uncountably infinite. Qualitative assessment, in the sense that

we want to generate plans that a human would reasonably devise, can only be convincingly done by looking at output.

To gather empirical evidence about the fire support planner’s performance, we implemented it in our WXXI combat simulation environment. This allows us to measure real world scalability and performance and to quickly (compared to hand simulation) generate automated fire support plans with 3D representations for face validation. The development of WXXI (the other components besides the fire support planner) may itself be considered a minor contribution since the research community now has at its disposal a relatively simple, open-source,³¹ discrete event-based combat model residing in the user-friendly Unity development platform with

- An HTN-based automated planning framework
- A fire support planning component
- A suppression model
- A data collection capability

The current version is available on the NPS GitLab repository at <https://gitlab.nps.edu/brharder1/wombatxxi>. In addition to the availability of the software, the implementation supports our understanding of the Conceptual Planning Framework and provides a case study for further refinement.

4. Evaluating the Implementation

To test the implementation, we use a four-phase process: three testing phases and a model development phase (Section VI.A.1). Our high-level description of this process may prove useful for researchers interested in extending this type of work. The reader should note that the non-exhaustive functional testing focuses on the most novel aspects of the research; a production system would require a more thorough testing plan.

Our experimental scalability results show that the prototype runs fast enough to support use as a scenario development tool, and perhaps even as a run-time dynamic

³¹ The current version of WXXI includes some paid assets from the Unity Asset Store. A user with the appropriate licenses receives full source code access to all of these assets in addition to the unique elements created for the WXXI project.

subroutine for small scenarios. There is a clear jump in running time from seconds to minutes when we scale up to the battalion-versus-company scenario. A few minutes of waiting is rather insignificant compared to the amount of time it would take for a human designer to generate a fire support plan of similar scale. However, the prospect of online planning, if this is desired, would demand better performance—in this case, the fire support planner would need to run at least once for every replication. There is certainly room for more efficiency in the algorithm.

Based on both quantitative and qualitative assessment, the fire support planner works well enough to support hasty scenario development. In situations where the generated plans are subject to careful human scrutiny, human quality assurance would be advisable. The system has the potential of supporting the creative process in a few ways: it could generate tactics that a human planner might not consider, and it could “fill in the gaps” by generating the remainder of a fire support plan that has been partially specified. Even without improving on some of its drawbacks, the system already demonstrates its usefulness for modelers interested in creating fire support plans for tactical scenarios.

The structure of the supported maneuver plan appears to be one of the main contributors to holding the fire support plan together. In the future, if maneuver planning is automated as well, the final resulting plan may yet appear haphazard and inhuman. Implementation appears to be the only way to evaluate this convincingly.

5. Implications

Automated tools such as the fire support planner and other proposed Conceptual Planning Framework components can fundamentally change the way that combat simulations are used in support of analysis and training. When scenario designers can rely on automation to generate reasonable tactical plans (or parts of them), they can raise the focus of their efforts by at least one level of abstraction. When a reasonable tactical plan for a scenario can be quickly generated, we can either speed up the M&S process or broaden the scope of the event.

Reliable online replanning should be the ultimate goal of near-term automated battle planning research. A static plan, no matter how well designed, becomes at least

partially ineffectual once unpredictable combat outcomes play out. Since all real world commanders change their plans to account for the changing situation, there is an argument that all simulations that do not perform online replanning are behaviorally invalid. The unacceptable behavior resulting from a static plan can be hidden from us, particularly in the analytical simulation domain, when we work with many replications and do not have the time to examine each one carefully. In real-time simulations (such as most training applications), the problem manifests as a need for many “white cell” human operators. In some cases, dynamic behavior rules can overcome the faults in a static plan, but actual human commanders use forward reasoning to predict outcomes, including the decisions and actions of the enemy—a feature that is not typically available to dynamic behavior models. Forward reasoning becomes increasingly important at higher echelons of command, where nearly all available actions have long lead times. We have only taken a few steps toward online replanning here, but the effectiveness of the fire support planner and its running time for small- to medium-size scenarios are promising results.

An executable combat model with a reasonably effective and efficient online replanning capability would have a significant impact on our approach to simulation. For analysis, scenarios could not only be generated faster—their results would be closer to what we would expect in the real world. No longer would scenario designers be limited to the kinds of tactical situations that lend themselves to a static plan. Furthermore, an ABPS could automatically account for experimental variable settings and generate a different plan for each design point—just as a real human commander would design a plan with the current capabilities and environment in mind. Since the planning system would use the same algorithms in each replication (with varying input), we avoid questions about the “fairness” of static plans. Identical static plans for different design points are not “fair” if they artificially favor one design point over the other, and creating a new static plan for each design point is an insurmountable burden unless we limit ourselves to just a few of them. Such a limited experiment does not take advantage of the space-filling experimental design advances of the last 10–20 years. Fairness may also be endangered by subtle confirmation bias on the part of the scenario designers or the tactics review board, but relegating tactics to an algorithm could help avoid bias.

An analytical study taking advantage of automated online replanning could work in the following way:

- (1) Formulate a hypothesis in terms of an experimental variable. This step is no different from the current approach.
- (2) Design the experiment, including the selection of experimental factors and design points. This is similar to the current approach, but we can now broaden the type and range of factors. All elements of the Planning Input (see Chapter III) are considered as possible factors, including the terrain map, Friendly and Enemy Force sizes and initial unit locations, and Tactics Configurations. Since we can depend on the ABPS, we use a space-filling design that is limited by computational power, not by scenario design time (although some manual effort will be needed to select unit starting locations and required tasks).
- (3) Perform pilot testing to validate that the planning system is able to generate reasonable plans for a wide range of starting conditions, including both initial planning and online replanning. This step may require adjustment of Tactics Configurations. Ideally, we use multiple tactical experts for the tactical validation.
- (4) Gather experimental data across the full range of design points. This step is no different from current approach, but we expect more data due to the space-filling design.
- (5) Analyze results. In theory, this is the same as the current approach—but now we have a richer data set to investigate.

We may find that it becomes *more* difficult to detect statistical significance in the experimental variable (chosen in step 1) once we begin varying other factors. If so, then interactions between factors may become the important statistics. This could offer more insight about the original research question than a traditional combat simulation study. For example, perhaps a new concept vehicle is only beneficial in certain kinds of terrain, or a new weapon is not any better than current weapons when using certain tactics. If even the interactions cannot detect a statistical difference, then we should be willing to retain the null hypothesis. For example, a small change to the maximum range of a weapon system might cause the automated planner to make slight changes to its plans—as human commanders would—such that there is no measurable benefit across the range

of realistic terrain and environmental conditions chosen for the study. This would be important information for an acquisition or doctrine decision.

In the real-time simulation domain, online replanning would reduce the number of human operators needed to provide tactical decision making input. For some applications, we would have the option of full automation for the aggressor force and all friendly units besides the one the user is controlling. Although this would not replace human-versus-human training, it would allow individual users to exercise tactical decision making and military science at any time and place, rather than waiting for a complex training event.

If online planning is dependable and fast enough for real-time training applications, then it may be viable for real world operational planning support. Although simulation is no substitute for the human mind in operational design and wargaming, being able to quickly play out multiple scenarios could provide new insights to the planning staff. By playing the role of lower-echelon commanders (those not controlled by human participants) for both the Friendly Force and the Enemy Force during wargaming, a combat model augmented with an ABPS could provide more rigor to the planning process, alerting planners to options and possible outcomes they may have not predicted.

The success of an ABPS depends on the users' trust in its capability and on its ability to improve the efficiency of M&S processes. The quantitative and qualitative results produced in this project should help in the trust-building process, but hands-on user feedback is an essential next step. If the video game AI community has anything to teach the combat modeling community about automation, it seems to be that designers need a certain amount of control over the behavior in their scenarios. This is evidenced by the fact that behavior trees are still the prevailing tool in computer games, as opposed to planning or learning techniques despite all of the progress and continued research in those areas. Although we would expect a simulationist to be comfortable with some unexpected results, we often need a certain kind of behavior to meet the objectives of a study or training event. With this in mind, it is important to include the configurability of tactics as we continue to develop automated planning for combat models.

C. FUTURE WORK

The Conceptual Planning Framework described in Chapter III is an abstract organizational construct, and as such cannot be empirically evaluated. Like many software engineering concepts, its usefulness can be argued subjectively and through case studies, but to date we have only a single, limited implementation from which to judge it. To improve the M&S community's ability to prepare, execute, and evaluate scenarios, the Framework must be further implemented and improved in realistic operating conditions. Follow-on projects could explore areas such as new derived models using sophisticated methods from the field of operations research; a richer implementation of planning styles and tactics configurations; new Mission Planners and Enhancement Planners focusing on functions such as maneuver, sustainment, reconnaissance, or communications; application of evolutionary planning techniques for Enhancement Planners; more complex designs for the Planning Controller; and a module for subdividing the Map into areas of operations.

Improvements to the theoretical data structures and algorithms of the Fire Support Planner are possible. The Extensions section of Chapter IV offers a few ideas. Researchers may consider new objective functions as alternatives to the risk value (i.e. expected losses), perhaps looking for nondecreasing cost and admissible heuristic (g - and h -cost) functions that would permit the application of A* search to fire support planning. Alternative search techniques such as beam search, or heuristics for limiting the number of planning branches, are worth exploring. Alternatively, one might apply the risk value to other domains as either a cost or heuristic function. The negative issues discussed in the Qualitative Results of Chapter VI provide more possible lines of effort. It would also be interesting to relax somewhat the assumption that defenders do not move, or to estimate the maximum useful forward planning time in an uncertain environment.

The implementation effort can be continued by either extending the WXXI-based ABPS or restarting the development effort in a production combat model. There appears to be a possible savings to the running time cost from conducting fire support planning in parallel at lower echelons. Increasing the autonomy of the maneuver planner using methods such as van der Sterren's HTN planner (2013) would improve the case for

adopting automated battle planning in the M&S community. With an eye toward an online replanning capability, it is important to develop techniques for detecting the need for plan repair, limiting the number and frequency of changes to plans, and maximizing the computational efficiency of planning—perhaps supported by a parallel computing architecture. Our intent is for WXXI to continue as an accessible research platform where new ideas can be quickly prototyped.

The strengths of the testing plan for this effort include the variety of maps, tactics, and force pairings; the statistical rigor applied to the quantitative results; and the mutually supporting assessment of quantitative and qualitative results. Expanding the testing approach is important for supporting further implementation of the Conceptual Planning Framework or improvements of the WXXI-based prototype. More rigorous and varied tests, such as human-versus-computer planning experiments, should help with the effort to expand adoption of automated battle planning tools. An expanded set of test scenarios with a richer set of unit templates, terrain types, and tactical approaches are bound to lead to more insights about the ABPS capabilities and reveal new opportunities for progress. An important value proposition for automated battle planning is the ability to speed up the scenario design effort and open the door for more space-filling designs of experiments for combat models. Demonstrating these kinds of robust experiments in a system like WXXI may catch the attention of both the operations research and the combat model development communities.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. COMBAT SIMULATION ENVIRONMENT IMPLEMENTATION

A. SYSTEM PURPOSE AND PRINCIPLES

Wombat XXI (WXXI) is a relatively simple ground combat simulation environment (CSE) developed as part of this research for demonstrating automated planning capabilities. It is built in the Unity 3D game development environment (Unity Technologies 2016), and therefore is written in the C# programming language. It resembles COMBAT XXI (CXXI) in several ways, but has a considerably smaller code base (not counting the source code of Unity) and less complex internal and external dependencies.

This appendix describes WXXI to a level of detail sufficient for understanding the research described in Chapters V and VI. It is not meant to be an operator's manual, and it does not cover the details of the ABPS described in Chapter V or Appendix B. Readers interested in viewing the source code can access it at <https://gitlab.nps.edu/brharder1/wombatxxi>. The project is not fully open source because it includes several modules procured from the Unity Asset Store. The source is freely available to anyone holding the appropriate software licenses for these modules. The author will provide any of the code actually written for this research upon request.

Formally, we should begin the development of an executable combat model by describing a simuland and corresponding referent, followed by the definition of a conceptual model. However, this research is focused on automated battle planning, not the development of an entire combat model. Therefore, we proceed directly to a description of the WXXI executable model.

Like CXXI, WXXI has entity-level resolution. This means that it maintains a separate object for each person or vehicle in the simulation (although the version used here has no vehicles). It includes a hierarchical unit organizational structure for both the friendly (attacker) and enemy (defender) side, and it has formation objects (collection classes) for moving groups of entities from one or more units together as a single

aggregate group. Each unit may be assigned a plan, which is a set of actions with scheduled start and end times. Actions instruct units to move along specific routes, fire on certain targets, or enable different behaviors. The simulation framework supports multiple replications of any number of different scenarios, and it writes out data for post-hoc analysis.

B. DISCRETE EVENT SYSTEM

Since the Unity run-time environment is optimized for high-performance graphics, the standard approach for behavior modeling is to run user-defined code (in the `MonoBehavior.Update()` function) between each graphical display frame.³² The graphical frame rate varies with system demand, so the behavior code cannot assume a fixed time step; it must account for the time since the last update and cannot precisely account for the duration until the next one. A fixed time step option is, in fact, available (the `MonoBehavior.FixedUpdate()` function), but is designed such that graphical frame updates can interrupt the fixed time updates. The graphical frame updates and fixed time updates interact in counter-intuitive and unpredictable ways. This is not usually a problem for real-time games, but it causes simulation results to become dependent upon the speed relative to real-time at which they are run and on the system load. This is a problem for testing, since we would prefer to execute many replications at the fastest available computation speed with outcomes dependent only upon a random seed and the experimental factors.

To work around this issue, we have implemented a discrete event simulation (DES) system through which all substantive combat events must pass. This logic resides in the `BasicEventProcessor` class, which uses the singleton pattern. Due to the way Unity `MonoBehavior` scripts work, an empty `GameObject` with a `BasicEventProcessor` Component must be placed in the scenario (Unity Scene). Rather than depend on the Unity time, we maintain a static variable in the discrete event system, which can be read with the call `BasicEventProcessor.CurrentTime`.

³² <https://docs.unity3d.com/Manual/EventFunctions.html>

Using an approach similar to that of Temiz (2016), `BasicEventProcessor` has two modes: Testing and Experimentation. Testing mode is useful for qualitative checks, and Experimentation mode maximizes processing speed for quantitative data collection. The other components of WXXI are ignorant of the processing mode; they always schedule events the same way and only request the current time from `BasicEventProcessor` (not from the Unity Time class). As a result, the outcome of any replication is independent of the graphical frame rate and computational resource competition, as desired.

In Testing mode, events are processed during the `MonoBehavior.FixedUpdate()` calls, but control is passed back to the game engine once `BasicEventProcessor` has caught up with the Unity time. Events seem to occur, from the human observer’s perspective, at whatever multiple of real-time speed is set in the Unity Time settings. If the system cannot achieve this speed, then time will appear to “slow down” to the maximum processing speed.

In Experimentation mode, events are always processed as fast as possible (which may be faster or slower than real-time), like a traditional DES. Graphics are only updated at a fixed rate with respect to simulation time—usually very sparsely.

Like any DES system, `BasicEventProcessor` maintains an event list ordered by scheduled simulation time. Other WXXI classes schedule events with the `BasicEventProcessor.waitDelay(EventCallback, float)` function. The `EventCallback` is a C# delegate invoked by `BasicEventProcessor` when the event fires. We borrow the `waitDelay()` name from the Simkit DES system (Buss 2011).

WXXI graphics prioritize qualitative analysis over aesthetics or image precision. We represent one entity firing on another with line segments that appear to travel from the source to the target. If these icons were displayed at proportional speeds—especially when running faster than real-time—they would appear in only one or two frames, if at all. This would not provide enough resolution for the human observer. Therefore, graphics are displayed with respect to real-time regardless of the rate at which `BasicEventProcessor` is processing events. Note that entity and unit locations are always

shown at their precise locations with respect to `BasicEventProcessor.CurrentTime`, but locations are only updated at the graphical frame rate.

C. TERRAIN

WXXI takes advantage of Unity Terrain objects³³ to represent the walkable surface. Unity Terrain data is stored as a two-dimensional array of elevation postings. The terrain is rendered as triangles with an edge between each laterally and longitudinally adjacent elevation posting and diagonal edges running southwest-to-northeast. The Terrain object includes length and width attributes that allows translation to the *xyz* Euclidean coordinate system of the rendering engine.

The terrain for all of our scenarios has a resolution similar to that of the military simulation systems we have in mind. The edges of the terrain triangles are 31m long or greater (large changes in elevation result in longer triangles).

Strictly speaking, any Unity Terrains—including handcrafted ones—can be used. Our scenarios use the Real World Terrain package (Infinity Code 2016) to import terrain elevation data from the Bing Maps database, which self-reports as accurate to 10 meters within the United States (where all of our scenarios are located). The Real World Terrain Helper (Figure 72) allows users to select a section of the world with a precise length, width, and location. The Helper provides this information to the Real World Terrain import tool, which generates the Terrain object in the Unity Scene. WXXI does not require the use of Real World Terrain; it only operates on the resulting Terrain objects, which are similar in structure to the CXXI terrain representation.

³³ <https://docs.unity3d.com/Manual/script-Terrain.html>

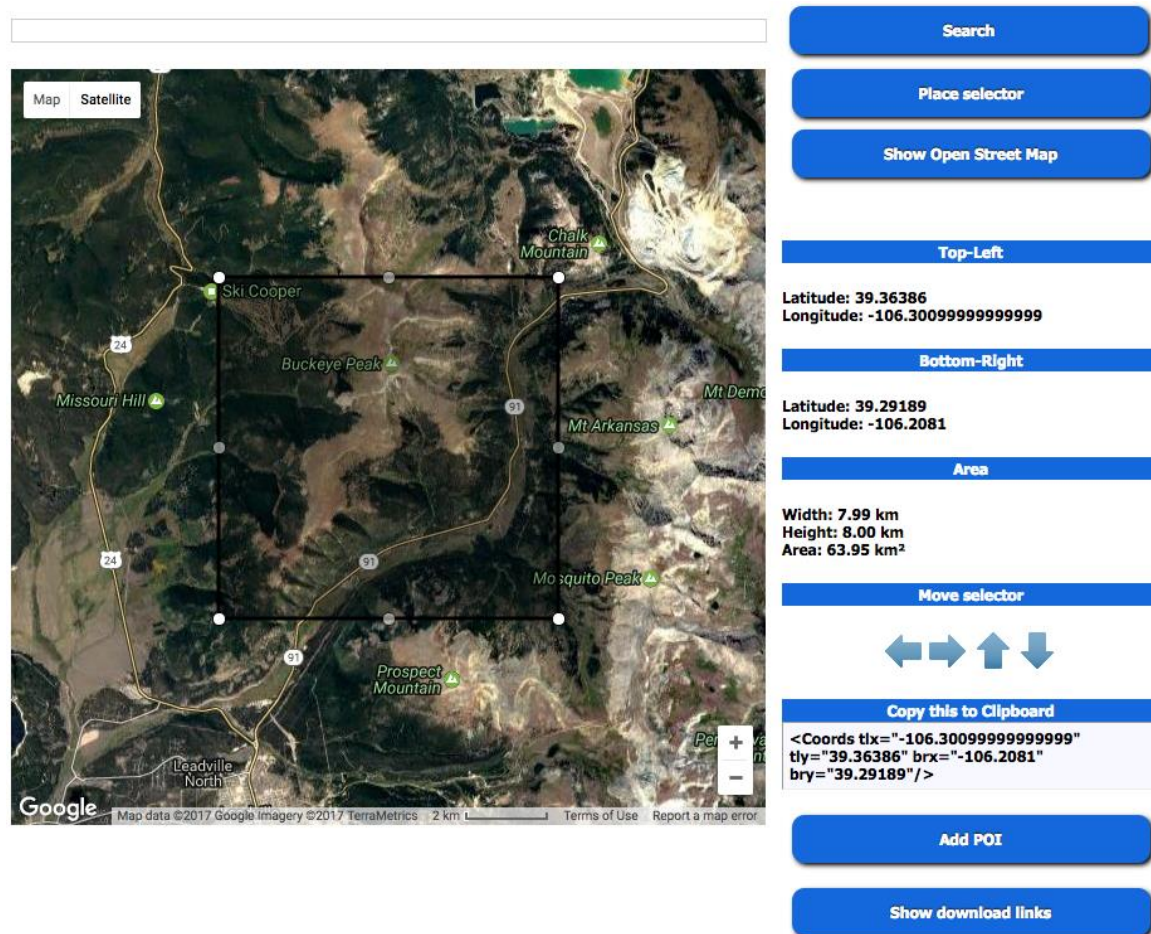


Figure 72. Real World Terrain Helper. Source: Infinity Code (2016)

The current version of WXXI does not use any obstacles besides the terrain skin. Objects such as trees, buildings, or vegetation areas could be added rather easily, since Unity is a fully featured game engine. Real World Terrain includes features to import roads, buildings, rivers, trees, and grass from map data, but we have not experimented with these.

D. ENTITIES

Entities in WXXI are represented by Unity GameObjects with custom Component objects (Figure 73), each derived from the Unity MonoBehaviour class (except the Capsule Collider, which is described in the Unity documentation). The Component separation is similar to the entity modeling approach of CXXI, but WXXI entities have

fewer components and a simpler architecture. In particular, there is no separate representation, other than ground truth data, for what entities perceive. The TargetSensor class, described in more detail below, provides references to the ground-truth GameObjects that it “detects,” and it has no “misclassification” model. This kind of fidelity did not appear useful for the work done here, but could be added to WXXI with new and updated Components.

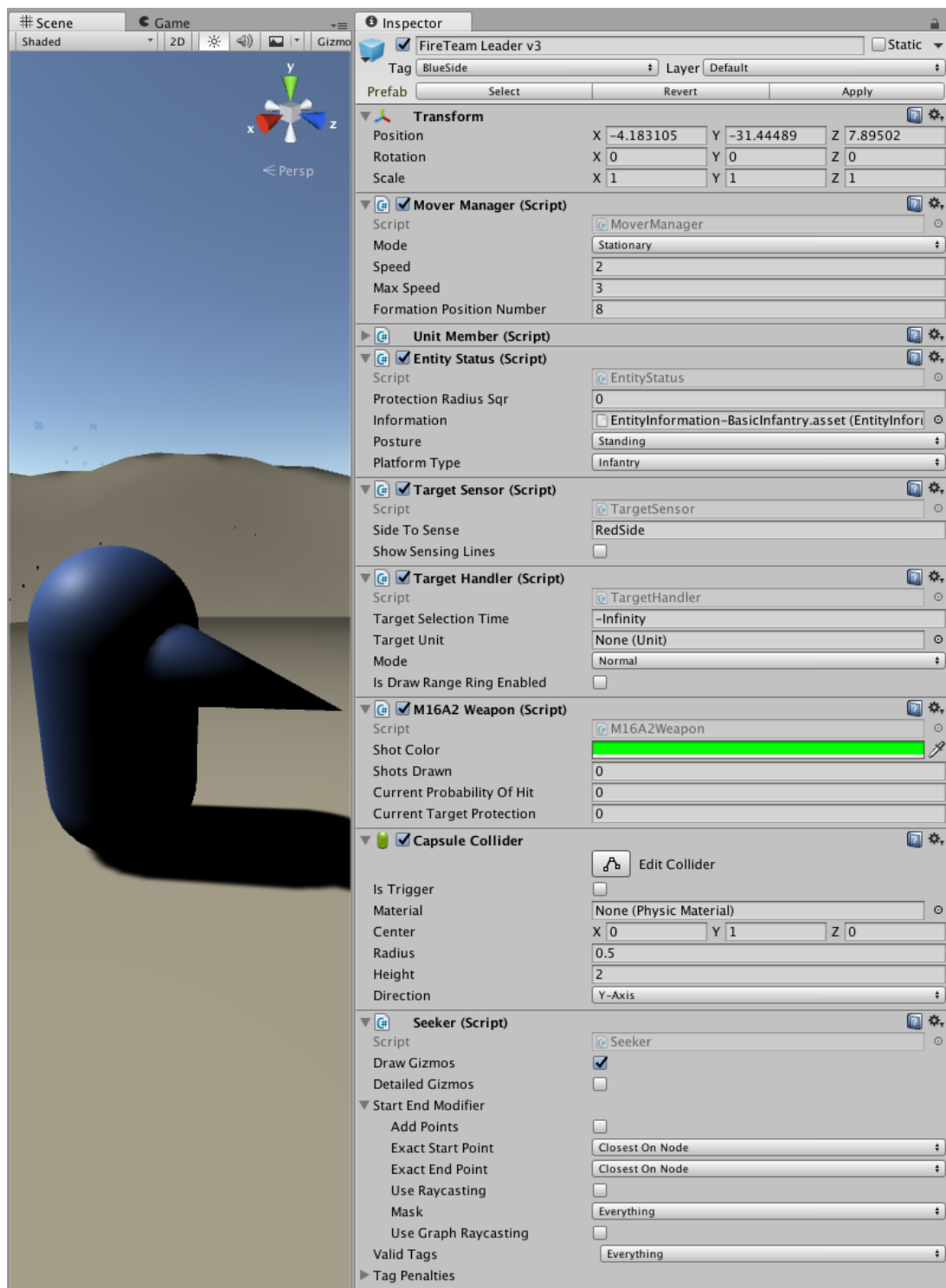


Figure 73. WXXI Entity GameObject

We describe the salient features of each custom Component in the following subsections.

1. Mover Manager

Movement in the current version of WXXI is modeled with a fixed time step “shoehorned” into the DES system. MoverManager contains the per-step movement logic. Every entity’s MoverManager.Start() function schedules a Move() callback on the BasicEventProcessor at the beginning of the scenario. Each Move() callback schedules another Move() one fixed time step in the future.

The potential problems with fixed time step movement are well known. The user may err on the side of precision by using a short time step; we use 0.1 simulation seconds for the experiments described in Chapter VI. This incurs a computational cost that would be avoidable (given some assumptions) with a true DES movement model. Upgrading WXXI to use DES movement would require more than just a change to MoverManager; we would need to replace all entity location queries with function calls to calculate positions with respect to time.

Each MoverManager can be set into one of the following modes to cause different movement behavior.

- stationary: no movement
- moveToDestination: the MoverManager of a Formation leader may receive a queue of waypoints (Vector3 objects) from its OrderMove() function. In moveToDestination mode, the entity moves towards the waypoint at the head of the queue. When it arrives within a certain radius of the waypoint (determined by its Formation) or starts getting farther away from it (indicating an overshoot), it dequeues the current waypoint and starts moving towards the next one.
- moveInFormation: MoverManager ignores its waypoint queue and moves towards its FormationTargetPosition, which is set by its Formation
- moveAggressively: MoverManager follows moveInFormation behavior until the entity’s TargetHandler component selects a target in one of its Formation’s TargetUnits. If so, MoverManager causes the entity to travel directly toward the target until reaching a minimum “close combat” distance. This is meant to allow the entity to get inside the target’s

protection radius (see EntityState below). This mode is used for entities engaged in an assault.

- pathfinding: MoverManager does nothing if this mode is set. This is a placeholder to allow a separate pathfinding Component to override MoverManager. It is currently not used.

Entities move at a speed determined by their current Formation, subject to a configurable maxSpeed.

MoverManager clamps its entity to the walkable surface using its GroundCast() function, which includes a layer mask parameter to model obstacles other than the Unity Terrain. WXXI has *no* entity collision model—the CapsuleColliders are only used for line of sight determination. In general, entities can pass through each other, but the moveAggressively mode keeps opposing entities slightly apart from each other to avoid degenerate targeting conditions.

2. Seeker

Seeker is a class defined in the Astar Pathfinding Project (Granberg 2016). Seekers are placed on unit leaders and are used by the ABPS at the start of a replication. They are, in fact, designed to be used during runtime and could be used to compute new waypoint queues for Formation leaders' MoverManagers with very little additional code. However, such an upgrade would need to deal with the loss of a formation leader to enemy fire. Options include placing a Seeker on every entity or transitioning the Seeker to the newly elected Formation leader as needed.

3. TargetHandler

TargetHandler determines which threat to engage, if any. The TargetHandler bootstraps itself into the event system in the same way as MoverManager, but it uses a stochastic “scanning and aiming” time between targeting events. This time is a combination of the entity's Weapon.getShotDelay() and its EntityState.getShotDelay(). The latter allows EntityState to increase time between shots when the entity is suppressed.

TargetHandler relies on the entity's TargetSensor Component to determine which targets are valid, i.e., within a maximum distance and visible. TargetHandler's job is to choose one of these and initiate firing on it.

TargetHandler has two modes:

- normal: engage the valid target that this entity has the best chance of killing, using the ground-truth probability of hit and probability of kill for its Weapon and the targets' attributes. If TargetHandler already selected a target in a previous callback, it will continue to engage that target until it becomes invalid, is killed, or after a maximum reselection time has passed. This models how troops tend to fire at a single target for a while before scanning for a new one; it also reduces the number of calls needed to TargetSensor.
- suppressing: randomly choose one of the valid targets from the TargetHandler's current targetUnit, if any. This mode is necessary for executing a fire support mission reliably.

In the current version, TargetHandler does not contain logic to reason about how long a potential target has remained in view or whether it is about to pass behind an obstacle. As an extreme but illustrative example, a target might be detected, immediately pass out of view for the entire waiting time, and then pass into view a microsecond before the next TargetHandler callback. This situation would be no different, as far as TargetHandler could tell, from a target that had been in view the whole time.

The fixed-time-step movement model obviates the need for a true DES approach to target detection, which would require more sophisticated equations to predict when targets will enter or exit each sensor's maximum range or pass in and out of cover. These become even more complicated if sensors and targets can both be mobile. In fact, no production combat models use this approach. Even CXXI, which follows a DES paradigm, uses a time step approach to target sensing—albeit a more sophisticated one.

4. TargetSensor

TargetSensor represents the eyes, binocular-aided eyes, thermal scopes, etc., of the entity, similar to the sensor objects of CXXI. WXXI currently allows only one TargetSensor per entity. The user must ensure that the SideToSense field has the correct

Unity tag string for the opposing side (e.g., BlueSide entity TargetSensors should be set to “RedSide”).

TargetSensor includes the targeting algorithms, but not the attribute settings (such as maximum range), for the entity’s sensor. The latter are found in its EntityInformation object, which is accessed through the EntityState Component (see below).

TargetSensor uses raycasting to determine which of the opposing side entities (or a given subset of them) are immediately visible and within sensing range. It can provide this list unordered or sorted from closest to farthest. Each targeting ray is cast from the entity’s TargetingOrigin to the potential target’s DetectionPoint; both points are configured as an offset from the entity’s base location coordinate in EntityInformation.

Entities block line of sight, which is a significant consideration for tightly grouped formations. For example, a unit arranged in a strict defensive line may find it difficult to employ its full firepower against a flanking opponent. However, TargetSensor is not restricted to a field of view or field of regard, unlike CXXI and some other entity-level models. Since WXXI only has a binary terrain occlusion model—no probabilistic or “fuzzy” concealment such as vegetation or rubble—TargetSensor is deterministic.

A variety of enhancements is possible within the TargetSensor code alone. Upgrading to a true DES detection model would require changes to TargetSensor and MoverManager as well, at minimum.

5. Weapon

Each entity may have one Weapon Component. Weapon is an abstract subclass of MonoBehavior. Currently, the only subclass of Weapon is the abstract class DirectFireWeapon. (Of course, this leaves the door open for an indirect fire class in a future upgrade.) DirectFireWeapon has a concrete subclass, M16A2Weapon, which is the Weapon assigned to most entities. We also have an abstract MachineGun subclass of DirectFireWeapon to support bursts of fire on a single target. Its one concrete subclass is M240MachineGun. The maximum ranges and cycle rates of both concrete Weapon classes are taken from publicly available sources on the M16A2 rifle and M240 machine

gun, but their accuracy and lethality attributes are arbitrary. We determined settings for which an attacker with 3:1 odds against a dug-in defender stands a very low chance of success, but a fully suppressed defender is very likely to be overrun.

DirectFireWeapons do nothing until TargetHandler calls their fire(GameObject) function, which provides the target entity as the only parameter. This function immediately determines whether the shot will hit or miss using the range-dependent getProbHit() function. The probability of hit is multiplied by 0.5 if the firing entity's EntityState is in the partiallySuppressed state, or by 0.25 if fullySuppressed. An additional penalty is assessed if the target is moving. We do not account for entities passing behind cover or making “lucky” turns during projectile flight; the result depends only on whether TargetSensor selected the target at the moment of firing and on the random draw against the getProbHit() return value.

If a hit is scored, Weapon schedules a hit() callback on the target's EntityState Component, passing itself as the parameter. Regardless of the hit or miss result, Weapon schedules suppress() callbacks—not just on the target, but also on nearby entities. The suppressionWeight value provided to each suppress() callback is scaled by the closeness of the previous random draw against the probability of hit (if a hit was scored, then suppressionWeight is maximal) and by the distance of the entity from the intended target. Both the hit() and the suppress() callbacks are scheduled after the computed time of flight, based on the range to the target.

DirectFireWeapons have four different qualitative engagement distances, which determine the *base probability of hit*, an initial value that will be multiplied by the probHitCoefficient() to determine the return value for getProbHit().

- Point blank: a target at distance 0 from the entity is hit with base probability BestProbHit. Targets are almost never at a distance of 0 (precisely collocated), but this concept is useful for describing the forthcoming calculations.
- ShortRange: entities that are moving, i.e., were displaced by a nonzero distance during their last MoverManager.Move() callback, may only engage targets at less than the ShortRange distance. This lets us model a severely restricted targeting capability due to an unstable firing platform, such as infantry walking across rough ground. The probability of hitting a

target at *exactly* the ShortRange distance, while moving, is 0. Targets at less than ShortRange have a nonzero probability of hit. ShortRange is not used if the entity is not moving.

- MediumRange: entities that are not moving have a base probability of hit equal to MediumRangeProbHit if the target is at *exactly* the MediumRange distance.
- MaxEffectiveRange: entities that are not moving have a base probability of hit of 0 if the target is at *exactly* MaxEffectiveRange or greater.

If the entity is moving, the base probability of hit is interpolated linearly between BestProbHit and 0, based on the actual distance divided by ShortRange. If the entity is not moving, we interpolate between BestProbHit and MediumRangeProbHit for targets within MediumRange, and between MediumRangeProbHit and 0 for targets between MediumRange and MaxEffectiveRange. This allows us to approximate a parabolic or hyperbolic probability of hit with a piecewise linear function (for nonmoving entities). The changing decay rate at MediumRange represents the “knee” of the curve.

The Weapon class also has a getProbKillGivenHit() function, which can take either a specific target entity, or a range and PlatformType, as its parameters. This function returns the probability of the target being killed under the assumption that it was hit. All special factors, such as the target’s cover and mobility, are already accounted for in the getProbHit() function, so getProbKillGivenHit() is quite simple. In the current version, both concrete weapon types just return a fixed kill probability of 0.5 per hit.

6. EntityStatus

The EntityStatus Component deals with fighting posture, suppression events, and casualties, and it provides a link to the configuration details in EntityInformation. EntityInformation is described in the next section.

EntityStatus has three posture states: standing, prone, and dugIn. Its getProtectionFrom(GameObject) function returns the protection score against a given entity. In the current version, this just returns the constant standing or dugIn protection score of its EntityInformation.

The suppression model is a three-state system. `EntityStatus` maintains a `suppressionTotal` value that is increased by each call to its `suppress()` function. Each `suppress()` call schedules an `unSuppress()` event after a fixed amount of “cool-off” time, which will reduce `suppressionTotal` by the same amount it was increased. When an entity’s suppression weight total reaches certain threshold values, it transitions from the unsuppressed state to `partiallySuppressed` and then `fullySuppressed`, where its rate of fire and probability of hit are penalized (as described in the Weapon section, above). The three-state model is analogous to that of CXXI; but in that system, scenario designers must write their own suppression effects such as the rate of fire and accuracy modifiers (Harder, Balogh, and Darken 2016).

When the `EntityStatus.hit()` function is called, a random draw against the source `Weapon.getProbKillGivenHit()` value determines whether the entity is killed. If so, `hit()` calls the `kill()` function. In WXXI, entities do not have a “health score” (they can be killed with a single hit), and there are no casualty types other than a kill. The `kill()` function puts `EntityStatus` into the dead state, removes the entity from any Formations and the Unit to which it belongs, and deactivates its `GameObject`. The `MoverManager`, `Formation`, and `UnitMember` classes handle election of a new unit leader or formation leader in the event that the killed entity held one of these positions.

7. EntityInformation

`EntityInformation` derives from Unity’s `ScriptableObject` class, so each instance is shared by all entities that reference it in their `EntityStatus` Component. We use only one such instance for the scenarios described in Chapter VI because all of the infantry entities on both sides are envisioned to have similar training, experience, and equipment. `EntityInformation` provides the following data:

- Platform Type: infantry is currently the only option, but future versions could include armor, motorized, helicopter, etc.
- Invincible: if true, this type of entity cannot be killed (useful for testing)
- Suppressable: if false, this type of entity cannot be suppressed (useful for testing)

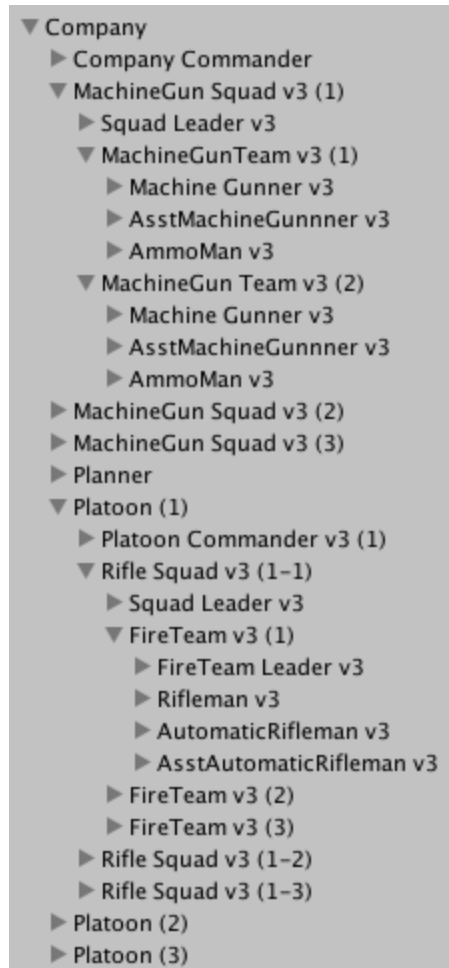
- Partial Suppression Threshold: the suppressionTotal at which this type of entity becomes partially suppressed
- Full Suppression Threshold: the suppressionTotal at which this type of entity becomes fully suppressed
- Suppression Cooldown Time: the scheduling delay used in EntityState.suppress() to schedule the unSuppress() callback
- Standing Protection: a value from 0 (no protection) to 1 (invincible) affecting the probability of hitting this type of entity while in the standing posture state
- Prone Protection: a value from 0 (no protection) to 1 (invincible) affecting the probability of hitting this type of entity while in the prone posture state
- Dug In Protection: a value from 0 (no protection) to 1 (invincible) affecting the probability of hitting this type of entity while in the dugIn posture state
- Protection Radius: entities closer than this distance use the Standing Protection value when attempting to hit this entity, regardless of its posture state—they are assumed to be inside the protective cover boundary
- Travel Speed: the normal traveling speed of this type of entity (a suggestion for the Formation speed)
- Max Speed: the entity may not move faster than this (can be used as a Formation catch-up speed)
- Traversable Surface Mask: a Unity layer mask indicating which types of labeled surfaces this type of entity can travel through
- Sensor Offset: a Vector3 added to the entity location (in local space) to determine its TargetingOrigin
- Detection Point Offset: a Vector3 added to the entity location (in local space) to determine its DetectionPoint
- Weapon Offset: a Vector3 added to the entity location (in local space) to determine the source of its Weapon's projectile flight path
- Primary Sensor Range: used by TargetSensor to filter out entities that are too far away to be detected

8. UnitMember

The UnitMember class serves as a link to the Unit containing each entity. It is used by the Unit class to find members. UnitMember's only functionality, besides reporting its owning Unit, is to remove its entity from that Unit when required.

E. UNITS, COMMANDS, AND PLANS

WXXI uses the Unity Transform Hierarchy as a user interface for organizing units and commands (Figure 74). Transform hierarchies are graphical tools with some features that are not appropriate for hierarchical military units—in particular, the translation of one transform moves all child transforms in exactly the same way. This can be useful to arrange large commands during scenario design, but it is not reasonable during a replication because each entity is supposed to move as an independent agent. Therefore, at the start of a replication, we disconnect the transforms of all units and entities from each other and replace those connections with references in the Unit and UnitMember classes.



In this company command structure, we show one machine gun squad, which contains two machine gun teams, fully expanded to the entity level. We also show one fire team of one squad of one platoon expanded to the entity level. The unexpanded units have symmetric organizations.

Figure 74. A Company Command Organization in the Transform Hierarchy

The Unit class represents both unit information (which entities it contains) and command information (its commanding unit and subordinate units). It can provide a collection of its own entities with the Unit.EntityMembers property or a collection of all entities in its entire command with the Unit.EntityMembersRecursive property. Subordinate units are reported with the corresponding Unit.SubUnits and Unit.SubUnitsRecursive properties.

The Unit.Remove(UnitMember) function, which is called by UnitMember, performs the obvious removal operation on its argument. If this entity is the unit's

commander, `Remove()` attempts to promote a remaining member of the unit to the commander position. If no members remain, it attempts to promote the commander of a `SubUnit` after using a recursive `Remove()` to pull that commander from its previous `Unit`. If `SubUnits` is empty, then the `Unit` is deactivated and considered destroyed. This promotion process models the succession of command used in real world military units.

`Unit` also holds an optional reference to a `Planner`, which it is responsible for invoking. In the current version, it only does this once, at time 0.0. Each `Unit` may have a `BattlePlan`, which is a set of `OperationalTasks` and `OperationalActions` for that unit alone. Without a `BattlePlan`, a `Unit`'s entities just stand in one place and fire reactively—which is exactly what the static defenders in our experiments do. A `Unit` object does not need a `Planner` to have a `BattlePlan`—one could be provided by a commanding `Unit`'s `Planner` or, in theory, manually created and loaded from file. `BattlePlan` and `OperationalTask` derive from `Unity`'s `ScriptableObject` class, which provides a facility for saving and loading instances. However, we have only tested plan generation through the ABPS.

F. FORMATIONS

`OperationalTasks` and `OperationalActions` are assigned to `Units`, but `Formations` are needed to move entities together. `Formations` are actually assigned at the entity level, but the `Unit.Formation` property reports the `Formation` that a `Unit`'s entities are in by returning a reference to its commanding entity's `Formation`. Multiple `Units` may be assigned into a single `Formation`—they do not even have to be part of the same command. The current use of this feature is to move squads (with subordinate team-sized units) and platoons (with subordinate squad-sized units) with a single instruction, without having to manipulate the subordinate units individually.

Each `Formation` object is essentially a container of entities with some movement functions. These include:

- `AssumePositions()` and `setTargetPositions()`: set the `FormationTargetPosition` of each `Formation` member's `MoverManager` based on the `Formation` leader's position. This is done periodically, and each time the `Formation` leader makes a turn, to ensure that the `Formation` maintains the correct approximate shape during movement.

- OrderMove(): instructs the Formation leader (and therefore the whole Formation) to move along a given path at a designated speed (for the leader) and catch-up speed (for the other members). By default, the members' MoverManagers use the moveInFormation mode, but OrderMove() can accept an alternate mode such as moveAggressively.
- OrderStop(): all entities stop immediately in their current positions.
- TargetUnits: the set of units the members of this Formation is allowed to engage. This is a suggestion for TargetHandler.
- WaypointTriggerProximity: a suggestion to the leader's MoverManager of how close it needs to be to the current waypoint to consider it successfully reached.
- CatchUpTriggerProximity: a suggestion to members' MoverManagers of how far from their FormationTargetPositions they should be before moving at the catch-up speed.

MoverManager allows an entity to be assigned to a JuniorFormation and a SeniorFormation, which provides a framework for formations-within-formations, a feature of CXXI. However, the types of formations used in this work do not use this capability. We only use the JuniorFormation property. Each entity may only be a leader of one Formation, which must be its JuniorFormation. The SeniorFormation is actually a property of the JuniorFormation object.

The loss of a subordinate unit within a Formation does not break the Formation. Formations automatically elect new leaders when the current leader is removed.

The Formations used in this work are

- RangerFileFormation: the simplest form of a tactical column, a so-called "ranger file" means each entity follows the entity in front of it. This allows the Formation to "snake" around turns, so each entity can take advantage of the same cover along a single path. Our implementation puts the leader in the front of the file.
- OnLineFormation: used in an assault, an "on line" formation means that each entity is shoulder-to-shoulder (with some dispersion between each) and facing the same direction. We place the leader in the center and keep the number of members balanced on either side.
- SupportByFireFormation: associated with a SupportByFirePosition, this Formation arranges entities in a static area. The positions are determined

by the spawning `SupportByFirePosition`, which is described in the next section. Entities in a `SupportByFireFormation` do not adjust positions upon taking casualties, since doing so would result in a temporary loss of suppressive firepower.

G. TACTICAL CONTROL MEASURES

The `TacticalControlMeasure` class represents the conceptual military planning objects of the same name. Each `TacticalControlMeasure` is a geometric object that can be placed on the terrain and used in `OperationalTasks` and `OperationalActions`. All `TacticalControlMeasures` can determine, based on entity locations, the following:

- `PartiallyOccupyingUnits`: the set of units with at least one entity inside the `TacticalControlMeasure`'s geometry
- `OccupyingUnits`: the set of units whose entities are all contained within the `TacticalControlMeasure`'s geometry

Subclasses of `TacticalControlMeasure` may have special fields and functions, as required by their semantics. WXXI includes the following `TacticalControlMeasures`:

- `AxisOfMovement`: a big blue arrow (literally) used to mark the start and end point of a `MoveByRouteAction`
- `AxisOfAttack`: a specialized `AxisOfMovement` used for an `AssaultAction`
- `StaticPosition`: an abstract subclass of `TacticalControlMeasure`. It defines a `positions(int)` property, which provides an array of n entity positions within its geometry (given n); and a `ChangeFormation(Formation)` function to transition its occupying `Formation` into a new kind of `Formation`, if needed.
- `StaticCircularPosition`: an abstract subclass of `StaticPosition` whose geometry is a circular area in xz space projected down onto the terrain
- `AttackPosition`: a `StaticCircularPosition` used in an `AssaultTask` to mark where a platoon should break into separate elements (i.e., a release point)
- `AssaultPosition`: a `StaticCircularPosition` used in an `AssaultTask` to mark where units should assume their assault formations
- `AssaultObjective`: a `StaticCircularPosition` for the objective area of an `AssaultTask`
- `SupportByFirePosition`: a `StaticPosition` whose geometry is a rectangle oriented toward a target position and projected down onto the terrain. In

the current version, its `positions()` function provides an evenly spaced line of positions, but this could be modified without affecting other components' source code. `SupportByFirePosition's ChangeFormation()` function returns a `SupportByFireFormation`.

H. ACTIONS

Since WXXI was developed with an ABPS in mind, it uses the same representation for operational actions (in the ABPS) and CSE actions. `OperationalTask`, which derives from Unity's `ScriptableObject` class, is the abstract superclass of all tasks and actions. `OperationalAction` is an abstract subclass of `OperationalTask` (this relationship is not a requirement of the Conceptual Planning Framework, but it proved economical for WXXI). By definition, operational tasks have no effect in the CSE; however, in WXXI concrete `OperationalTasks` (that are not `OperationalActions`) may be part of a `BattlePlan`. They do nothing but report their starting and ending events in the console. `OperationalActions` are allowed to issue state-changing commands to entities, units, and formations.

`OperationalTask` has `StartTime` and `EndTime` properties. These are used by `BattlePlan` to schedule each `OperationalTask's` `Begin()` and `End()` functions. `OperationalActions` may override `Begin()` and `End()` to provide their state-changing code. `Begin()` may also schedule additional events prior to `EndTime` if needed, but generally speaking, periodic effects should be handled by entity-level behaviors (`MoverManager`, `TargetHandler`, etc.). Nothing restricts the `OperationalActions` of a unit from running concurrently, but all `OperationalActions` in each `BattlePlan` are totally ordered for our experiments. `OperationalTasks` often overlap because they represent multiple subordinate tasks or actions.

`OperationalActions` may be instantaneous. If so, the `Begin()` and `End()` functions are scheduled at the same time and the `End()` function has no effect. If we think of `OperationalActions` as elements of a continuous timeline specification (Section C.6 of Chapter II), then instantaneous `OperationalActions` are analogous to events, and non-instantaneous `OperationalActions` are analogous to persistence conditions.

WXXI does not currently use the persistence plan extension discussed in Chapter II (Section C.9), but this could be implemented by writing `OperationalActions` with conditional checks in their `End()` functions. If the check fails, `End()` can re-schedule itself at some point in the future. `BattlePlan` can handle subsequent tasks that start late as a result of this, but will skip tasks whose `EndTimes` have been surpassed before they have started. A robust implementation of persistence plans would require a more adaptive `BattlePlan`, or a replanning capability, to deal with delays.

Important parameters (represented as instance properties) and functions of `OperationalTask` (and `OperationalAction`) are listed here. Other properties and functions used only in the ABPS are described in Chapter V and Appendix B.

- `Unit`: the `Unit` object tasked by this `OperationalTask`
- `EnemyUnit`: the `Unit` targeted by the tasked `Unit` (or null if not applicable)
- `SerialNumber`: a unique integer for this `OperationalTask` instance

I. RUN MANAGERS AND DATA COLLECTION

WXXI's `BasicRunManager` and `MultipleRunManager` classes support quantitative experimentation. To write out the results of a replication to file, the scenario (Unity Scene) must have a `BasicRunManager` object, which can be placed on an otherwise empty `GameObject` (usually named `RunManager`). A software component may create a new “row” of data by calling `BasicRunManager.NewDataItem()`. The return value is of type `GDEBCMRunDataData`, an automatically generated class from the Game Data Editor asset (Stay at Home Devs 2016). After the replication has ended, `BasicEventProcessor` calls `RunManager.EndSimulation()`, which appends the contents of the data item to a comma-separated-value text file.

The data fields include the following:

- `IsFireSupportPlanningEnabled`: Boolean. Was the `FireSupportPlanner` enabled in this replication?
- `TerrainResolution`: the number of horizontal elevation postings

- `MaximumFireSupportTaskCountPerUnit`: each unit's `BattlePlan` was limited to this maximum number of `FireSupportTasks` (m from the complexity analysis of Chapter IV)
- `FireSupportPlannerIterations`: the number of iterations the `FireSupportPlanner` completed before outputting the plan
- `MinimumFireSupportTaskDuration`: `FireSupportTasks` of duration less than this amount (in simulation time) were discarded and not considered for the plan
- `TimeSpentInitializingFireSupportPlanning`: see Chapter VI
- `TimeSpentInFireSupportPlanningLoop`: real-time spent in the main loop of `PlanFireSupport()`
- `TimeSpentInPlanFireSupport`: see Chapter VI
- `TimeSpentScanningNavGraphs`: see Chapter VI
- `TimeSpentAnnotatingNavGraphs`: see Chapter VI
- `TimeSpentPathfindingForFireSupport`: see Chapter VI
- `TimeSpentPlanningManeuver`: real-time spent for maneuver planning
- `TotalPlanningTime`: real-time spent for both maneuver planning and fire support planning
- `SimulatedRunningTime`: the simulation time point at which the replication ended
- `ActualRunningTime`: the real-time elapsed during the replication, not counting planning time
- `UnsuppressedShotRate`: measured average rate of fire of `RedSide` entities that had a valid target and were in the `unSuppressed` state
- `PartiallySuppressedShotRate`: measured average rate of fire of `RedSide` entities that had a valid target and were in the `partiallySuppressed` state
- `FullySuppressedShotRate`: measured average rate of fire of `RedSide` entities that had a valid target and were in the `fullySuppressed` state
- `DefenderOverAttackerFER`: see Chapter VI
- `MissionAccomplishmentScore`: see Chapter VI
- `CreationDateTime`: time stamp for the data item

MultipleRunManager handles multiple replications of multiple scenarios by taking advantage of Unity's Scene management features. A MultipleRunManager must be placed on a GameObject in an empty Unity Scene (one not containing any WXXI scenario information). Its Unity Inspector display contains most of the required input (Figure 75). The user provides the number of different scenarios (the Size field), the filename of every scenario (Scene) that needs to be run (Scenario Files Element fields), the number of replications per scenario, and the name of the output file to be written by BasicRunManager after each replication.

Multiple Run Manager (Script)	
Script	MultipleRunManager
▼ Scenario Files	
Size	24
Element 0	CayucosCreek_Plt-vs-Squad_Disabled
Element 1	CayucosCreek_Plt-vs-Squad
Element 2	PaloAltoVA_Plt-vs-Squad_Disabled
Element 3	PaloAltoVA_Plt-vs-Squad
Element 4	BuckeyePeakCO_Plt-vs-Squad_Disabled
Element 5	BuckeyePeakCO_Plt-vs-Squad
Element 6	PaloAltoVA_Co-vs-Plt_Disabled
Element 7	PaloAltoVA_Co-vs-Plt_Best
Element 8	PaloAltoVA_Co-vs-Plt
Element 9	BuckeyePeakCO_Co-vs-Plt_Disabled
Element 10	BuckeyePeakCO_Co-vs-Plt_Best
Element 11	BuckeyePeakCO_Co-vs-Plt
Element 12	BuckeyePeakCO_Bn-vs-Co_Disabled
Element 13	BuckeyePeakCO_Bn-vs-Co_Fastest
Element 14	BuckeyePeakCO_Bn-vs-Co_Faster
Element 15	BuckeyePeakCO_Bn-vs-Co_Best
Element 16	BuckeyePeakCO_Bn-vs-Co
Element 17	CayucosCreek_Plt-vs-Squad_Close
Element 18	PaloAltoVA_Plt-vs-Squad_Close
Element 19	BuckeyePeakCO_Plt-vs-Squad_Close
Element 20	PaloAltoVA_Co-vs-Plt_Best_Close
Element 21	BuckeyePeakCO_Co-vs-Plt_Best_Close
Element 22	BuckeyePeakCO_Bn-vs-Co_Best_Close
Element 23	DarkRangePeak_SmallSandboxScenario
Replications Per Scenario	1
Data File Name	gde_data_bcm

Figure 75. Multiple Run Manager Inspector Pane

All of the scenario names in MultipleRunManager's Scenario Files must also appear in the Unity Project's "Scenes In Build" field (Figure 76), found at File / Build Settings. This window supports drag-and-drop from the Project window. Once this setup is complete, a single click of the Play button results in all scenarios running sequentially, in the order listed under Scenario Files, each for the same number of replications. (Useful tricks here include enabling Edit / Project Settings / Player / Settings for PC, Mac & Linux Standalone / Resolution / Run In Background, and turning off Gizmos to speed up execution.)

At the end of each replication, BasicRunManager writes out its data, then finds the MultipleRunManager and invokes its EndReplication() function. This allows MultipleRunManager to determine the next step or quit.

Currently, the only way to get slight variations of a particular scenario is to create multiple scenarios. Copying existing scenarios can speed up this process. Also, note that the Planner in each scenario runs at the start of every replication, generating the same plan (per scenario) multiple times. This approach allows multiple measurements of the Planner's running time for scalability testing, but if only combat performance results are needed then the BattlePlans or final PartialPlan could be saved and recycled, since they derive from ScriptableObject. This option has not been tested and may require some modifications to deal with lost references to instance variables.

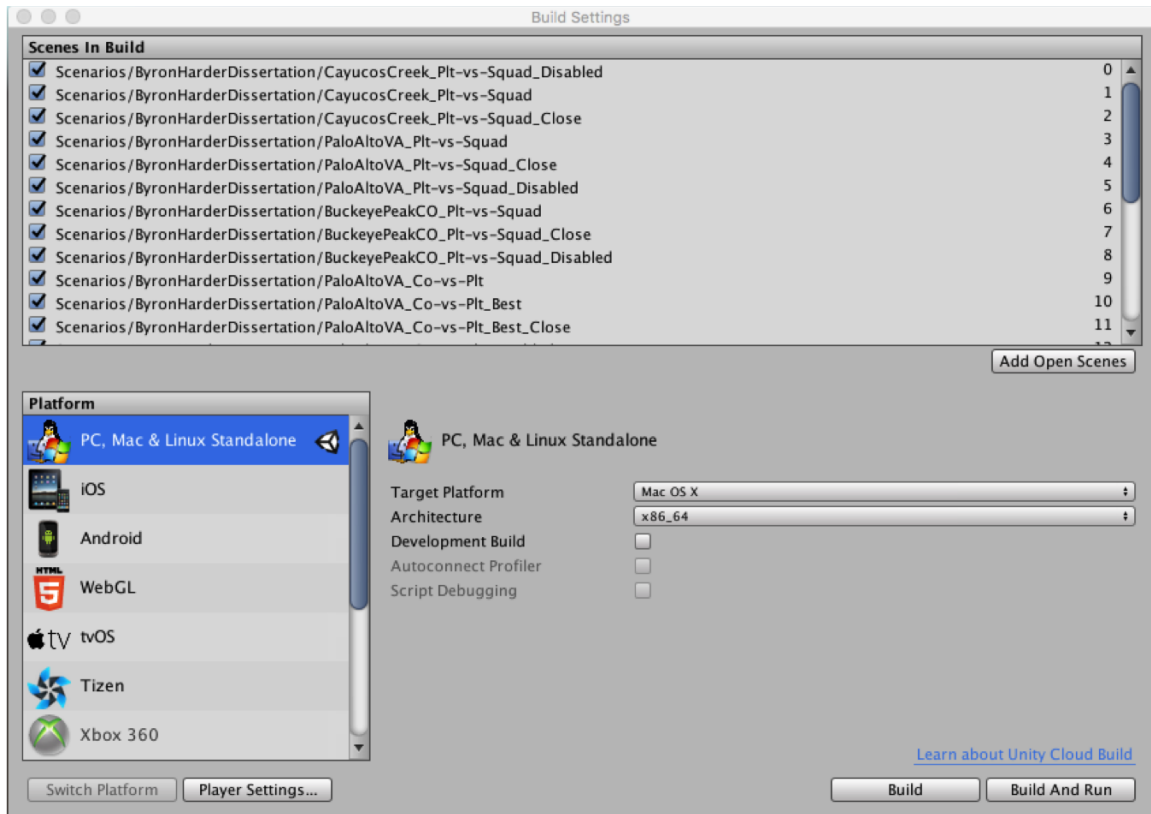


Figure 76. Unity Build Settings

APPENDIX B. AUTOMATED BATTLE PLANNING SYSTEM DETAILS

A. OVERVIEW

This appendix provides additional details for some of the modules in the fire support planner implementation. It should be useful for researchers interested in extending the ABPS implementation or migrating its capabilities to a different CSE.

B. VISIBILITY GRAPH EDITOR

The inputs to the `VisibilityGraphEditor` are

- Character Radius: the radius of a cylinder large enough to contain any entity considered for movement (see Pathfinding in Chapter V)
- Max Slope: the maximum grade that entities are allowed to ascend or descend (see Pathfinding in Chapter V)
- Terrain: the scenario's Terrain object (see Annotated Mobility Graph in Chapter V)
- Threat Unit: the commanding unit of the Enemy Force (see Enemy Force in Chapter V)
- Friendly Unit: the sample unit to be used for mobility calculations (see Annotated Mobility Graph in Chapter V)
- Visibility Height: the maximum entity height (see Annotated Mobility Graph in Chapter V)
- Visibility Distance: the maximum targeting distance to be considered for threat annotations (see Annotated Mobility Graph in Chapter V)
- Max Subdivisions: for future use; set to -1 (see Annotated Mobility Graph in Chapter V)
- Penalty Weight: a scaling factor to set the importance of risk versus speed in pathfinding searches (see Pathfinding in Chapter V)
- Show mesh outline and Show node connections: for user visualization only; no effect on the planning system
- Initial Penalty: not used; set to 0. This is a standard feature of the Astar Pathfinding Project. The value is a constant added to any node penalty

computations (which are described in the Annotated Mobility Path section of Chapter V).

C. TASK NODE FUNCTIONS AND FIELDS

TaskNodes are the elements of the partial plan representation, which is formatted as an HTN. TaskNode properties and member functions include

- (1) TaskNode.OperationalTask or TaskNode<T>.Task: the OperationalTask contained in the TaskNode.
- (2) Children: the set of TaskNodes that fall immediately below this TaskNode in the plan hierarchy due to decomposition of this TaskNode. Children should not include a member of the Ancestor set (see below) or the TaskNode itself since this would cause a loop and break the hierarchy. The time interval (from StartTime to EndTime) of each child's OperationalTask should be contained in the closed time interval of this TaskNode's OperationalTask.
- (3) Descendants: the closure set of all Children, all Children of those Children, and so on.
- (4) Parent: the TaskNode immediately above this TaskNode in the hierarchy. A child TaskNode is created when its Parent is decomposed.
- (5) Ancestors: the closure set of the Parent, the Parent of the Parent, and so on.
- (6) Successors: the set of all TaskNodes explicitly ordered after this TaskNode. This TaskNode's OperationalTask should complete before any of its Successors begin. Successors do not require any hierarchical relationship to this TaskNode. In fact, it would not make sense to have an Ancestor or Descendant as a Successor since each TaskNode's time interval is bounded by its Ancestors' time intervals. A *sibling* TaskNode—one that has the same Parent—may be assigned as a Successor. A support relationship can be modeled with Successor relationships to other parts of the hierarchy.³⁴
- (7) Predecessors: the set of all TaskNodes that have this TaskNode in their Successors set.

³⁴ Support relationships are better supported with a richer set of temporal relationships. For example, a logistical support task and the supported maneuver task should both start at the same time and run concurrently. Without a “during” relationship, we could introduce other tasks such as “PrepareToSupport” with two Successors: the supported maneuver task and a “ProvideSupport” task.

- (8) **RiskIntervals**: the set of RiskIntervals associated with this TaskNode and all Descendants. This is returned as an enumeration to prevent the caller from adding or removing elements (although they could create a new RiskIntervalSet using references to or copies of the RiskInterval objects in the enumeration). Only OperationalActions can build a RiskIntervalSet, and TaskNode<OperationalAction> objects cannot have Children, so every RiskInterval is associated with a leaf node. The set of RiskIntervals for any non-leaf node is actually a set of references to the RiskIntervals of the leaf nodes below it in the hierarchy.
- (9) **RisksFromThreatUnit(Unit threat)**: an enumeration of RiskIntervals whose ThreatUnit properties match the threat parameter.
- (10) **RiskValue**: the risk value of this TaskNode's RiskIntervals (including all Descendants)
- (11) **IsFullyGround**: currently, this returns true if the TaskNode's OperationalTask is an OperationalAction and false otherwise. A more thorough implementation would check whether all of the OperationalTask's parameters are set with non-null values (except those that are allowed to be null according to the semantics). IsFullyGround is not used directly by FireSupportPlanner.
- (12) **IsAtomic**: this returns true if the TaskNode has no children (is a leaf node) and false otherwise.
- (13) **ShallowCopy()**: returns a new TaskNode with a copy of the original TaskNode's OperationalTask and no hierarchical (Parent, Children, Ancestors, Descendants) or temporal (Successors, Predecessors) relationships. If the original TaskNode's IsAtomic property is true, then the TaskNode copy receives a copy of the original TaskNode's RiskIntervalSet. Otherwise, it has an empty RiskIntervalSet.
- (14) **DeepCopy()**: returns a copy of the entire tree rooted at the original TaskNode. The original TaskNode and all of its Descendants are copied with ShallowCopy(), and new, identical hierarchical and temporal relationships are set among the copied TaskNodes with one exception: the Parent of the returned TaskNode is left as null. The RiskIntervals of the non-leaf copied nodes are set to the union of RiskIntervalSets of its leaf-node Descendants *after* the leaf nodes have received their copied RiskIntervalSets, so nothing in the new copy refers to anything in the original.
- (15) **ForEach(TaskNodeDelegate del)**: applies del to this TaskNode and all of its Descendants.

- (16) `ForEachAtomicTask(TaskNodeDelegate del)`: applies `del` to this `TaskNode` if it is a leaf node. Otherwise, applies `del` to all Descendants that are leaf nodes.
- (17) `Ordered(ICollection<TaskNode> originalCollection)`: this static function returns an array of `TaskNode` references in nondecreasing order of `OperationalTask StartTime`s.

D. PARTIAL PLANS

The `PartialPlan` class has a reference to a root `TaskNode`. All of the `PartialPlan`'s details are contained in the `TaskNode` hierarchy at and below the root. Useful member properties and functions include the following:

- (1) `DecomposableTasksSnapshot`: an enumeration of all `TaskNodes` that can be decomposed—that is, leaf nodes that are not `OperationalActions`. *Snapshot* means that the enumeration members do not change even if some of them are decomposed while enumerating.
- (2) `DecomposeExpand(TaskNode decomposingNode, IEnumerable<TaskNode> newNodes)`: `newNodes` are added to `decomposingNode` as Children. Nothing is copied. Any Descendants of `newNodes` remain in place, and therefore become part of the `PartialPlan`. Temporal relationships are not modified by this function, but all relationships involving `decomposingNode` implicitly affect its new Descendants. This function should be used only if a single planning branch is being maintained (i.e., greedy search) because it does not create a copy of `decomposingNode` or any other `TaskNodes` in the `PartialPlan`.
- (3) `DecomposeSwap(TaskNode decomposingNode, TaskNode newNode)`: `decomposingNode` is removed from the `PartialPlan` and replaced with `newNode`. All of `newNode`'s Descendants remain and therefore become part of the `PartialPlan` as well. Any temporal relationships set on `newNode` are erased, and all temporal relationships of `decomposingNode` are copied to `newNode`. Temporal relationships of `newNode`'s Descendants are unaffected, but the relationships applied to `newNode` implicitly affect its Descendants after the decomposition. This function should be used only if a single planning branch is being maintained (i.e., greedy search) because it does not create a copy of any of the other `TaskNodes` in the `PartialPlan`.
- (4) `DecomposeCopy(TaskNode decomposingNode, TaskNode newNode)`: rather than modify the calling `PartialPlan`, this function returns a new `PartialPlan`. The new `PartialPlan` is created by first copying the original plan, and then calling `DecomposeSwap(decomposingNodeCopy,`

newNode), where decomposingNodeCopy is the copy of decomposingNode in the new PartialPlan.

- (5) ImmediateSuccessors(TaskNode taskNode): this function returns all of the explicit and implicit one-step Successors of taskNode in the PartialPlan. Explicit Successors are simply the members of taskNode's Successors set. Implicit Successors are the Descendants of all Explicit Successors, the Successors of taskNode's Ancestors, and the Descendants of the Successors of taskNode's Ancestors.
- (6) GenerateGameObjectRepresentation(GameObject prefab): displays a textual, hierarchical representation of the PartialPlan in the CSE for simulation developers or scenario designers.

E. METHODS

The HTNMethod class is the format for methods in this ABPS implementation. HTNMethods decompose TaskNodes to create branches for automated planning. The HTNMethod class includes the following parameters (fields) and functions.

- (1) name: a unique string identifying the method, similar to a class name.
- (2) types: a set of OperationalTask types to which this method may be applied—usually just one, but not limited to one. This allows a particular tactic to be used to accomplish several different tasks. For example, a frontal attack could be used to accomplish doctrinal tasks such as *destroy*, *seize*, and *clear*.
- (3) MeetsConstraints(PartialPlan partialPlan, TaskNode taskNode): uses the method's meetsConstraints delegate to determine whether this method can be applied to taskNode given the current state of partialPlan. This can include checking parameters of taskNode's contained OperationalTask.
- (4) Apply(TaskNode taskNode): uses the method's decomposer delegate to decompose taskNode. Returns a set of TaskNodes (usually with Children) that can each be used with a planning branch.

HTNMethod provides variants of Apply() with more restricted functionality:

- (5) ApplyFirst(TaskNode taskNode): only returns the first taskNode created by the decomposer delegate.
- (6) TryApply(PartialPlan partialPlan, TaskNode taskNode): combines the constraint check and the decomposer call in a single member function. If the constraint check fails, then an empty set is returned.

- (7) TryApplyFirst(PartialPlan partialPlan, TaskNode taskNode): a combination of ApplyFirst and TryApply, this returns the first decomposed TaskNode if the constraint check succeeds, and null otherwise.

F. RISK INTERVAL DATA STRUCTURES

1. Risk Sets

RiskIntervalSet objects are created by calling its static Build() functions, which have a few different signatures. Build() functions call RiskInterval constructors and support the following situations:

- A single defending Unit and a single attacking Unit following an IAnnotatedMobilityPath
- All relevant defending Units for a single attacking Unit following an IAnnotatedMobilityPath
- All relevant defending Units for a single attacking Unit in a single location on a given IAnnotatedMobilityGraph
- All relevant defending Units for a single attacking Unit in a single location in a given IAnnotatedMobilityNode

The RiskIntervalSet class includes the following functions and properties.

- (1) RiskValue: the sum of the RiskValues of its member RiskIntervals
- (2) NullSubset: the RiskIntervalSet containing only the members whose RiskValue is 0.0 (or less)
- (3) StartTime: the earliest StartTime of any member
- (4) EndTime: the latest EndTime of any member
- (5) Threats: an enumeration of all unique threat Units of its members
- (6) RemoveNullifiedRiskIntervals: deletes from this collection all RiskIntervals whose RiskValue is 0.0

2. Risk Intervals

RiskInterval offers the following important properties and functions:

- (1) StartTime: the StartTime of its earliest RiskIntervalSubregion

- (2) `EndTime`: the `EndTime` of its latest `RiskIntervalSubregion`
- (3) `StartLocation`: the `StartLocation` of its earliest `RiskIntervalSubregion`
- (4) `EndLocation`: the `EndLocation` of its latest `RiskIntervalSubregion`
- (5) `RiskValue`: the risk value of this `RiskInterval`, which is the sum of the `RiskValues` of its `RiskIntervalSubregions`
- (6) `FriendlyUnit`: the `RiskInterval` represents expected losses against this attacking Unit
- (7) `ThreatUnit`: the `RiskInterval` only reflects expected losses due to this defending Unit
- (8) `ThreatUnitSize`: the number of entities in `ThreatUnit` at the time the `RiskInterval` was constructed
- (9) `SerialNumber`: a unique integer to identify this `RiskInterval`. This is useful for error checking, since we end up with many copies of each `RiskInterval`
- (10) `BuildFromPath(Unit enemy, Unit friendly, EntityInformation entityInformation, IAnnotatedMobilityPath annotatedPath, int startIndex, int numberOfNodes)`: this constructor-type static function is used to create a new `RiskInterval` from a nontrivial path (with more than one waypoint). It takes advantage of `IAnnotatedMobilityPath`'s `NodeVectorPaths` property to determine endpoints for risk segments due to turns in the path and node boundaries. It creates additional segment endpoints using the `RangeCrossingPoints()` function of the threat Unit's `TargetHandler`, which is described below. These points are used to create the `RiskInterval`'s single initial `RiskIntervalSubregion` (which contains the risk segments).
- (11) `AccountForTask(OperationalTask task, float startEffectsTime, float endEffectsTime)`: subdivides the `RiskInterval` at `startEffectsTime` and `endEffectsTime` (see `TrySubdivide()`) and then calls `ApplyMultiplier()` on each `RiskIntervalSubregion` bounded by those time points.
- (12) `TrimNullifiedRisk()`: removes `RiskIntervalSubregions` from the beginning and end of the `RiskIntervalSubregions` list that have `RiskValue 0.0`
- (13) `ReplaceSubregionsFrom(RiskInterval other)`: used by `FireSupportTask.ReduceRisk`. Replaces the `RiskIntervalSubregions` with those of the parameter `RiskInterval`, by reference.
- (14) `TrySubdivide(float time, out RiskIntervalSubregion earlierSubregion, out RiskIntervalSubregion laterSubregion)`: subdivides a contained

RiskIntervalSubregion at the provided time unless there is already a subdivision at that exact time (see Section D.4 of Chapter IV)

3. Risk Subregions

RiskIntervalSubregion offers the following fields and functions:

- (1) StartTime: the start time of its first risk segment
- (2) EndTime: the end time of its last risk segment
- (3) StartLocation: the first waypoint
- (4) EndLocation: the last waypoint
- (5) RiskValue: the risk value of this RiskIntervalSubregion
- (6) ContainedWaypoints: the list of endpoints for its risk segments
- (7) SegmentLength: the list of distances between adjacent ContainedWaypoints. The i -th entry of SegmentLength is the distance from ContainedWaypoints[i] to ContainedWaypoints[$i+1$].
- (8) ThreatDistance (and ThreatDistanceSqr): the list of distances from the assumed threat location to each point in ContainedWaypoints (and their squared values)
- (9) MidPointThreatDistance: the list of distances from the assumed threat location to the midpoint of each risk segment
- (10) Speed (and SpeedSqr): the constant speed of the attacking unit assumed for time calculations (and its squared value)
- (11) WaypointTimes: the expected time for the attacking unit to reach each ContainedWaypoint
- (12) ThreatDirectionDotVelocityTimes2: as its name implies, the list of dot products $2\mathbf{x}_i \cdot \mathbf{v}_i$, where \mathbf{x}_i is the direction vector from the assumed threat location to ContainedWaypoints[i] and \mathbf{v}_i is the velocity vector for the friendly Unit's traversal from ContainedWaypoints[i] to ContainedWaypoints[$i+1$].
- (13) InitializeRiskValues(TargetHandler threatTargetHandler, EntityInformation friendlyEntityInformation): computes and stores risk values of each risk segment in this RiskIntervalSubregion by calling CalculateSegmentBaseRiskValue() on each adjacent pair of points in ContainedWaypoints.

- (14) `CalculateSegmentBaseRiskValue(int j, TargetHandler threatTargetHandler, EntityInformation entityInformation)`: calculates the risk value of a single risk segment by calling `RiskValueLinearIntegral` (see Section D.6.) on the threat's `TargetHandler`.
- (15) `ApplyMultiplier(double multiplier)`: multiplies the internal `riskMultiplier` and `RiskValue` by the parameter. Used when a new `FireSupportTask` is added to the `PartialPlan`.
- (16) `Subdivide(TargetHandler threatTargetHandler, EntityInformation friendlyEntityInformation, float subdivisionTime)`: breaks up a risk segment at `subdivisionTime`.

G. CONSTRUCTING THE MANEUVER PLAN

The `buildManeuverTask()` function is used to construct a maneuver plan using detailed user input. The starting position is a parameter of the `buildManeuverTask` function. The `AssaultPosition` (for platoon and squad assaults) and `AssaultObjective` (only for squad assaults) are taken from the tasked unit's `waypointRoute` parameter, which is set manually by the user. The user's selection of initial unit locations, `AssaultPositions`, and `AssaultObjectives` determine the maneuver tactics for the planner—in other words, the planner does not make tactical *maneuver* decisions (except for the detailed routes, described hereafter). The `buildManeuverTask()` function generates totally ordered child `TaskNodes` under the `MissionTask` containing the following `OperationalTasks`:

- A `ChangeFormationAction` to put the command (the tasked unit and all subordinate units) into a single column formation (`RangerFileFormation`)
- A `MoveByRouteAction` to move the formation to the `AssaultPosition`. The planner calls `IAnnotatedMobilityGraph.FindRoute()` to get a tactical path (balancing risk and speed) for the `MoveByRouteAction`.
- If the unit is a rifle platoon, a `MissionTask` for the assault of each subordinate rifle squad. This is done with a recursive call (one per squad) to `buildManeuverTask()`.
- If the unit is a rifle squad, a `ChangeFormationAction` to put the command (the squad leader and all subordinate fire team members) into a single `OnlineFormation`
- If the unit is a rifle squad, an `AssaultAction` to move from the `AssaultPosition` to the `AssaultObjective`. The planner uses

IAnnotatedMobilityGraph.FastRoute() to get the shortest possible path, ignoring the path's risk. The focus on speed results in a more aggressive assault.

When buildManeuverTask() is called on a rifle platoon, it replaces the MissionTask with an AssaultTask. As described above, the only platoon-level actions are to form the whole unit into column and move it to the platoon-level AssaultPosition. An assault is then planned for each squad with the platoon-level AssaultPosition as its starting position. Each squad must have a squad-level AssaultPosition and an AssaultObjective in its waypointRoute. If the user provides reasonable positions, the resulting plan has the following basic flow:

- The platoon moves together in column, trying to avoid threat observation, to the platoon assault position (which serves as a release point, in doctrinal terms)
- The squads break off and each move in column to their squad assault positions
- The squads form into online formations
- At approximately the same time, the squads assault their individual objectives by the most direct routes

H. POTENTIAL FIRE SUPPORT SETS

PotentialFireSupportSet includes the following properties and functions.

- (1) BestOption: returns the TaskNode<FireSupportTask> with no required movement with the best score, or if all require movement, the one with the best score. This corresponds to the determination of P_t and the argmax operation in Section F.2 of Chapter IV.
- (2) HasOptionsTargeting(RiskInterval riskInterval): although a FireSupportTask can reduce the risk from several RiskIntervals, PlanFireSupport creates each FireSupportTask with one particular RiskInterval as the primary target (which determines the values t_a and t_b in Algorithms 2 and 3). FireSupportTask has a property called TargetedRiskInterval to hold this piece of information, and HasOptionsTargeting determines whether any member of the PotentialFireSupportSet has the riskInterval parameter as its TargetedRiskInterval. This allows FireSupportPlanner() to avoid re-work when creating new potential tasks against new RiskIntervals.

- (3) `UpdateTaskOptions(TaskNode<FireSupportTask> selectedTaskNode)`: This function iterates through the `TaskNode<FireSupportTask>` objects in its collection and uses the `UpdateAffectedRiskValues()` and `StoreEffectsOn()` functions of the contained `FireSupportTasks` to modify the `RiskValues` of potential tasks, altering their scores. If a route was replaced in the most recent `PartialPlan` update, this function also removes the old route's `RiskIntervals` from any potential tasks that depended on them for their score calculations.

I. PLANNING-RELATED OPERATIONAL TASK FEATURES

The following properties and functions, defined in the `OperationalTask` class, are used for automated planning.

- (1) `IsRiskModifier`: returns false if this `OperationalTask` can never reduce the `RiskValue` of any `RiskIntervalSubregions`, and true otherwise
- (2) `RiskValueMultiplier(RiskIntervalSubregion)`: Returns the fractional amount of `RiskValue` that would remain for the given `RiskIntervalSubregion` if this `OperationalTask` were successfully executed. Defaults to 1 (no reduction).
- (3) `ReduceRisk(IEnumerable<RiskInterval>)`: reduces the `RiskValues` of the `RiskIntervalSubregions` of the given `RiskIntervals`, assuming this `OperationalTask` is successfully executed when planned
- (4) `AffectedTimeIntervals`: Returns a series of ordered and nonoverlapping [start, end] time pairs covering all of the time intervals that this task can have an effect on risk intervals. These are not necessarily bound by the start and end times of the task.

J. FIRE SUPPORT TASKS

`FireSupportTask` includes the following special-purpose member functions.

- (1) `StoreEffectsOn (OperationalTask childTask, IEnumerable<RiskInterval> riskIntervals)`: The `riskIntervals` parameter is assumed to contain only `RiskIntervals` whose `ThreatUnit` matches the target of this `FireSupportTask`. This function iterates through `riskIntervals`, copying and attempting to reduce the risk of each member using `childTask.ReduceRisk()`. When this results in an actual reduction, the original `RiskInterval` reference and its reduced copy are stored in `PriorAffectedRiskIntervals` and `PostAffectedRiskIntervals`, respectively. The `ReduceRisk()` function only works for `OperationalActions`, so `StoreEffectsOn` must be called with each risk-reducing child of each relevant `FireSupportTask`. In the current implementation, the child

SupportByFireAction (there is exactly one per FireSupportTask) is the only risk-reducing child, but more complicated task structures could be supported by this generalized code. See Section F.7 of Chapter IV for a pseudocode version of this function.

- (2) UpdateAffectedRiskValues(OperationalTask selectedChildTask): This simply calls selectedChildTask.ReduceRisk() on the PostAffectedRiskIntervals and AdditionalAffectedRiskIntervals. See Section F.6 of Chapter IV.
- (3) ReduceRisk(IEnumerable<RiskInterval> riskIntervals): FireSupportTask has a special implementation of ReduceRisk. Since it already has updated versions of all of the RiskIntervals it affects (constructed from copies of the RiskIntervals in the PartialPlan), there is no need to re-compute that same data. Instead, it uses the hash mappings from its PostAffectedRiskIntervals to its PriorAffectedRiskIntervals, the latter of which are all in the PartialPlan's RiskIntervalSet. For each of these, it simply replaces the collection of RiskIntervalSubregion objects in the PriorAffectedRiskIntervals member with its updated version in PostAffectedRiskIntervals. This leaves all of the RiskInterval objects in the PartialPlan's RiskIntervalSet in place—an important detail, since all of the other potential tasks have hash-maps to those same RiskIntervals.

APPENDIX C. MILITARY OPERATIONS AND PLANNING

A. OVERVIEW

We describe here a few of the military processes and methods relevant to battle planning and fire support. Since we propose to model military planning, at least in terms of input and output, this doctrine serves as the referent. An important refrain throughout this review is that human-centric processes are designed to take advantage of human military intuition and buttress the shortcomings of human reasoning—in other words, they provide little detail for the more challenging aspects of a computer-based implementation, and they do not assume the availability of computers’ strongest features. We assume that the intent for an ABPS is reasonable output, not fine-grained modeling of every step of the military planning processes. With this in mind, we begin by distinguishing between military concepts that are appropriate for computational modeling and those that, at least for today, are better left to human commanders and planning staffs.

B. TACTICS

Tactics is defined in doctrine as “the application of combat power to defeat the enemy” (DOD 1997b, 3) and “accomplish missions” (DOD 2008, 3), or “the employment and ordered arrangement of forces in relation to each other” (DOD 2011a, I-14). The Army and Marine Corps use this term in the full spectrum of operations between full-scale conventional warfare and stability operations, but our research is focused on conventional combat. Any discussion of tactics is based on the concept that there are many different ways a military force can be arranged and employed, some more effectively than others for a given situation and objective.

Since an ABPS is a model of human planning, its tactics should resemble those of a human planning team. For many applications, users are interested in tactics that adhere to a particular doctrine, but still with the implication that plans are the result of (modeled) human interpretation of that doctrine in the context of the situation at hand. This approaches the question of validation, which we consider in its own section below.

Generating valid tactics seems more within reach when we distinguish between *tactical art* and *tactical science*. The former speaks to the creative genius involved in overcoming a thinking opponent's will to fight, and the latter refers to the codified techniques and processes used to deploy and employ military forces (DOD 1997b, 3–5; DOD 2008, 1–2 to 1–3). The line between these concepts is often blurry. For example, employing deception to achieve surprise is an explicit element of maneuver warfare doctrine (DOD 1997a, 43–44), but a particularly clever and effective deception is clearly a stroke of military art. We argue that the current capabilities of automated battle planning have many shortfalls in terms of tactical science, and that the “creative genius” and human mimicry of AI systems in general is not mature enough to make tactical art a sensible goal for automation at this time. Our focus, therefore, is on improving the tactical science of automated battle planning. Unfortunately, art and science are often interwoven in military publications. Western-style doctrine has a stronger emphasis on tactical art than science, so we must be selective about what we draw upon from the planning referent.

Tactics is understood in military circles to be the most decisive factor in the outcome of a battle (MCDP 1–3, Foreword). Military history is replete with examples of inferior forces winning the day through superior tactics. For most analytical uses of M&S, we do not want the outcome of a replication to depend on the “genius” of a modeled commander since this would likely obscure the intended experimental effects. On the other hand, we do not want outcomes to be determined by tactically ignorant behavior either, such as units rushing across open ground when a concealed route is available or failing to account for the maximum effective range of weapons. *Reasonable* behavior is what most applications need—plans that account for the modeled environment, assets available, and known threats at a level commensurate with an average commander. In other words, we would prefer automated planning to be done intelligently, but not ingeniously. Certain specialized applications of automated battle planning, such as real world planning support or tactics training, could benefit from more surprising or insightful machine-generated maneuvers, but there is enough work before us without going that far.

C. ORGANIZATION OF FORCES

Military forces are organized through command and support relationships among their units. The most important and universal of these is the command hierarchy. The principle of *unity of command* states that every unit, and every individual unit member, follows the direction of a single commander (DOD 2011a). At the top headquarters of any command, the most senior commander is subordinate to some external commander or civilian authority. Since the commanders of each unit also respect the direction of their senior commanders, the entire command (collection of units) has a hierarchy of command relationships. Each level of this hierarchy is called an *echelon*. Military tradition provides a few special names for each echelon. For example, the echelon between platoon and battalion (or squadron) is called company, battery (for artillery), or troop (for cavalry). Each command includes some type of headquarters unit comprised of just an individual leader at the smallest echelons, a team-sized unit at intermediate echelons, up to headquarters units hundreds or thousands strong at the highest levels. The headquarters may itself be a command, containing its own hierarchy of units.

U.S. doctrine defines a few different types of command relationships: combatant command, operational control and tactical control. The difference has to do with the types of tasks that may be issued and the extent to which a commander may reorganize subordinates. These distinctions do not come into play in this work; the reader may assume an operational control relationship in all cases, if needed. The important point is that each unit is required to fulfill the tasks issued by its commanding headquarters.

Some military forces, such as multinational coalitions, do not fall under a single commander. In these situations, other relationships are established between the commanders of the disjoint commands. These tend to be unique to coalition agreements, but we can think of them as a class of *coalition relationships*.

In addition to command and coalition relationships, commanders may establish *support relationships* among subordinate commands. Each support relationship directs a *supporting command* to aid a *supported command*. Support relationships can be as complicated as needed, but the most basic elements are the *function* and *category*. The

function determines what types of tasks the supporting command is required to perform, as able. The category determines the process by which support tasks are communicated and answered (DOD 2013b, V-8 to V-10). Direct support, in which a supporting unit receives direct calls for support from a single supported command, is the most common category. Support relationships are relevant for both planning and execution. During execution, they determine which commands may call upon other commands not subordinate to them for assistance. Support relationships are relevant to planning for reasons such as the following:

- The supporting command may need to conduct online planning in order to find a way to fulfill a request.
- A support relationship may provide external resources for a supported command to consider as options during its own planning.
- A supporting command may need to fulfill requests from supported commands' plans during its own planning.
- A supporting command may need to generate tasks on its own initiative to aid in the accomplishment of the tasks in the supported unit's plan.
- A command may dictate changes to subordinates' support relationships as part of its plan.

D. LEVELS OF WARFARE

Tactical is the bottom tier of the three levels of warfare (DOD 2011a, I-12), which we depict in Table 10. The tactical level, compared to the upper levels, is distinguished by the importance of precise spatial and temporal reasoning and relatively short time scales. Smaller tactical commands focus on *engagements*—distinct episodes of combat lasting less than a day. The higher tactical echelons consider *battles*, which are comprised of multiple engagements (DOD 2012a, 1–1). Since some battles include only one engagement, and since “engagement planning” is not a common term, we use the term *battle planning* to refer to planning for one or more engagements.

In contrast to the tactical level, the *operational* level is distinguished by its grand geographic scale, episodic discretization of time, and greater focus on logistics and large-scale deployment of forces along lines of operation. The operational level deals with

whole series of battles grouped into major operations and campaigns. The highest level of warfare is *strategic*, including in its scope nonmilitary capabilities such as diplomacy and economic leverage (DOD 2011a, I-13).

Table 10. Levels of Warfare and the Echelons within Them

Level of Warfare	Echelon
Strategic	Nation
	Regional Theater
Operational	Joint Force or 3-star Command
Tactical	Division
	Brigade or Regiment
	Battalion or Squadron
	Company, Troop, or Battery
	Platoon
	Squad or Section
	Team

In Table 10, we subdivide each level of warfare into the ground unit echelons that typically comprise it. Although this subdivision is not prescriptive (in doctrine or the real world), most of the CSEs of interest fall clearly within the tactical level. (We describe several CSEs in Chapter II.) Our focus in this research is the tactical level, but Table 10 shows that this still covers a wide range of echelons. Lower echelons must deal more with pathfinding and geometry of fires. Higher echelons deal with coordination and geographical massing or economy of force, and they tend to have more varied assets to allocate (Pew and Mavor 1998, 207–215). Each successive echelon, moving upward in the table, tends to plan further into the future and use wider subdivisions of time. In real world commands, sibling subordinates are not always of the same echelon. For example, a Marine division includes three infantry regiments, an artillery regiment, a light armored reconnaissance battalion, and an armored battalion, so battalions and regiments can be siblings.

Doctrinal publications for specific types of units and battlefield functions tend to provide more information on tactical science than tactical art, making them more easily translatable into computational models. Often, these types of documents have terms such

as *tactics, techniques, procedures, field manual*, and (for the Marine Corps) *warfighting* in their titles. For example, ATP 3–21.11 provides the following guidance for Stryker companies upon reaching the probable line of deployment (PLD) during the conduct of an attack: “The attacking unit splits into one or more assault and support forces once it reaches the PLD, if not previously completed. All forces supporting the assault force should be in their support-by-fire positions before the assault force crosses the PLD” (DOD 2016a, 2–45).

E. OPERATIONS ORDERS AND PLANS

Joint doctrine distinguishes orders from plans. An *operations order* (OPORD) is issued to subordinate units for actual execution in the near future, but *contingency plans* are built for potential future conflicts, with many assumptions about the anticipated situation. Contingency plans are published to help subordinate units focus their training and other preparations, but they are less detailed than OPORDs and are not directive for execution (DOD 2011b, II-15 to II-25). The human processes for creating OPORDs and contingency plans are essentially identical, so we have no need to distinguish them here. Formal contingency plans are normally developed at only the strategic level. Since our focus is the tactical level, and since the AI community uses the term *plan* to refer to near-term, executable instructions in a similar sense to an OPORD, we use *plans* and *orders* (including *OPORD*) interchangeably.

In real world military operations, an OPORD is issued by each headquarters unit. Each OPORD provides taskings for its directly subordinated commands, but usually not for the subordinate commands of those subordinates. The details of how to carry out these taskings are left to the subordinate commands, which develop their own OPORDs for their own direct subordinates. Each subordinate headquarters finds its assigned tasks in the commanding unit’s OPORD and uses them as objectives for its own planning process; this continues recursively down to the smallest units. This approach creates tiers of abstraction spaces; each commander and staff need only reason at a few echelons (above, at, and below their own) and to issue tasks at their subordinates’ echelon. The complete set of instructions issued to all echelons of a large command, after all planning

is complete, is a collection of many OPORDs with the same hierarchical structure as the command's units.

Military planning doctrine straddles art and science just as tactics doctrine does. The essence of “planning art” is in the way commanders *change* plans as events unfold, and in designing plans that can be readily changed. The mission statement and commander's intent sections of an OPORD support changeability by essentially restating and describing in more detail the most critical outcomes required. The remaining parts of the OPORD, such as taskings and coordinating instructions, simplify the process of changing plans because they remain in effect unless specifically altered. During execution, changes occur with greater frequency at lower echelons. This protects the higher-echelon plans from the chaos and confusion of combat to some degree.

OPORDs for military units have a common doctrinal structure, called the *five-paragraph order*. We present here the five paragraphs and a few of the important subparagraphs.

1. Situation

In this section, we find background information not specific to the tactical approach of the plan. This includes current intelligence on enemy forces, the issuing command's area of operations, and the status of friendly forces outside the issuing command.

2. Mission

This section contains only the mission statement, mentioned earlier. Dedicating this statement to its own paragraph helps highlight its importance to the human reader.

3. Execution

The detailed tactics are contained in the execution paragraph.

- (1) Commander's Intent: as described above, this is the commander's description of the desired effects of the operations, meant to guide the decisions of subordinates when events do not proceed exactly as the plan predicts.

- (2) Concept of Operations: this provides a prosaic narrative of how the plan should unfold. This aids the understanding of human readers, but there is usually no information here that is not presented in other parts of the execution paragraph. The plan is usually described in terms of sequential phases and parallel efforts.
- (3) Task Organization: any reorganization of the command's subordinate units is explained here.
- (4) Tasks: the directive tasks for subordinate commands and staff sections are listed here. One command is designated as the main effort—the commander's bid for success. All other commands are supporting efforts. Support relationships are usually listed here as tasks for the supporting commands.
- (5) Coordinating Instructions: tactical control measures, execution matrices (for timing), and instructions common to multiple subordinate commands are found here.

4. Administration and Logistics

Administration is rarely modeled in CSEs, but resupply, fueling, ammunition management, and maintenance are sometimes factors. Logistical information is found in this paragraph or in a referenced annex.

5. Command and Signal

This section lists any deviations from the standard command relationships between the issuing command's headquarters units and its subordinate units. It includes the succession of command, in case the commander becomes a casualty, and the planned locations of headquarters units. Signal information includes radio and information network configurations as well as call signs and code words. This may be important for CSEs that model communication explicitly.

F. MILITARY PLANNING PROCESSES

Military doctrine includes two different planning processes: the troop leading steps for small units with no staff, and the formal planning process (which goes by several names) for larger units with dedicated staff sections.

1. The Troop Leading Steps

The troop leading steps focus on reconnaissance. Since small units are more greatly affected by geometry of fires, a commander gains a better understanding of what tactics may be useful by personally viewing the terrain. This process gives little detail about how to go about choosing and developing the actual plan. Its main purpose is to give less experienced leaders a structured way to gain insight about the problem at hand and disseminate information to subordinates. The troop leading steps are

- (1) **Begin Planning:** the commander receives orders from the senior headquarters and gathers all relevant information, including the mission, enemy, terrain and weather, troops and fire support, time, space, logistics, and civilians (known as METT-T or METT-TSL). To the extent possible, he issues a warning order to subordinate units to allow them to begin preparations.
- (2) **Arrange for Reconnaissance:** the commander assembles a reconnaissance team, including himself, and develops a mini-plan to conduct reconnaissance of the objective area or other key geography. If personal reconnaissance is not possible, the commander requests surveillance and intelligence support from external assets.
- (3) **Make Reconnaissance:** the commander carries out the reconnaissance plan.
- (4) **Complete the Plan:** using the information gained through reconnaissance, the commander finalizes the battle plan.
- (5) **Issue the Order:** the commander delivers the plan to the subordinate units.
- (6) **Supervise:** the commander ensures that subordinate units correctly prepare for the mission, responding to requests for clarification or concerns about assigned tasks.

Our intent is to focus on the planning activities after the military intelligence process (including reconnaissance) has completed. Therefore, the troop leading steps do not offer much insight about the decision making processes involved in planning.

2. The Formal Planning Process

Larger headquarters with dedicated staff follow a more involved planning process. Modern armed services in countries around the world use an almost identical

approach to planning that revolves around the course of action concept. On its surface, this process is a step-by-step method of developing a plan from a handful of options using the analytical power of a few human minds in a time-constrained environment. The most important outcome of real-world military planning is not a plan, though; it is a deeper understanding of the tactical problem for the commander and staff. This improves their ability to change the plan when predictions and assumptions become invalidated. The mental process of improving one's understanding of the situation in order to visualize and describe solutions is called *operational design*. Since current AI systems are not capable of the sentient thought required for operational design, the doctrinal planning processes are not necessarily the best framework for an automated planner. Nevertheless, the steps of the formal military planning process are part of the planning referent, so we review them here. The following generic process is a distillation of the Army's Military Decision making Process (MDMP) (DOD 2012b), the Marine Corps Planning Process (MCP) (DOD 2010b), Joint Operation Planning Process (JOPP) (DOD 2011b), and most other service and allied nation planning processes:

- (1) Problem Framing and Mission Analysis: the staff seeks all relevant information for the task at hand, including the initiation of military intelligence processes. Some intelligence may already be available from external assets. The planners examine their assigned area of operations on a map marked with mobility zones. This helps them visualize possible avenues of movement and think about tactical options. The staff carefully reviews the specified tasks from the higher headquarters OPORD, discerns implied tasks to make the specified tasks possible, and distills the most essential tasks into a concise mission statement. It sends requests for information up to higher headquarters as it finds data shortfalls and makes assumptions to allow planning to proceed.
- (2) Course of Action Generation: using experience and intuition, the staff proposes a handful (three is a common number) of distinct courses of action (COAs) for completing the mission. They sketch the key aspects of each COA using the subordinate commands as the moving parts. COAs may vary by task organization, forms of maneuver, axes of advance, and approaches to timing and coordination. The number of different COAs is determined in part by the amount of time available and the size of the planning staff. The staff proposes COAs with distinct features to support operational design.

- (3) Wargaming: the staff roleplay through COAs, with some playing the enemy side, to gain better insight about the strengths and weaknesses of each option. They record results for the upcoming COA decision, noting where predisposed notions about enemy actions appear to be wrong—an inevitable result of wargaming against a thinking opponent (the staff members playing the enemy's role). To improve the fidelity of the wargame, it is common to model (usually with rocks or grease pencils) commands two echelons below the staff's level, even though the OPORD will only task the immediate subordinates (one echelon below). The adjudication of engagements with the (wargamed) enemy is usually left to the judgment of the wargamers or a senior officer designated as a judge. Although doctrine admits the possibility of computer simulations in support of wargaming, in reality this is only done at the highest echelons.
- (4) Course of Action Decision: in this step, the staff compares the COAs against each other for the commander, who then chooses one—or a combination of more than one—to form the basis of the plan. This decision hinges on the tactical artistry and operational design of the commander more than quantitative analysis of wargaming results. The commander is not obligated to follow the recommendations of the staff or remain within the scope of any COA. If the commander has concerns about the command's ability to fulfill its assigned tasks, he communicates this to the senior commander.
- (5) Orders Development: with the selected course of action as a basis, the staff adds the required level of detail to the plan. Only enough detail to direct the actions of the next lower echelon is typically provided. This gives maximum flexibility to subordinate units in developing their plans.
- (6) Transition: the commander or lead planner delivers the order to all subordinate units. Subordinate units begin their own planning process using the tasks in the order as input. The staff (of the issuing unit) uses feedback from subordinate unit planning to refine the order and issues changes as needed. Units at all levels prepare for execution, to include the conduct of rehearsals.

3. Modeling the Planning Process

The troop leading steps and the formal planning process seem different on the surface, but in fact, they cover the same activities. The troop leading steps highlight the reconnaissance effort because personal reconnaissance is important for the operational design of a small unit leader, but reconnaissance and other military intelligence activities are also part of the formal planning process, particularly during the problem framing stage. The troop leading steps do not mention COAs, but the small unit leader should

consider different options before deciding on a plan. Issuing the order and supervising, from the troop leading steps, are equivalent to the formal planning process's orders development and transition stages. Due to the similarity of these two processes, it seems that a single, generic process is appropriate for a computational model. However, some parameters and code probably need to be specialized for each type of unit and each echelon.

An assumption of this research is that generating useful plans for simulated units is more important than representing the planning process. Since achieving human-quality operational design in a machine is not a reasonable goal, it seems more useful to pursue approaches that take advantage of a digital computer's strengths—precise, fast calculation. We expect *reasonable* plans to be achievable without modeling a commander's insight or interaction with a staff. Generating COAs, numerically ranking them by a combination of important factors, and choosing the one with the highest score seems to be a reasonable computational approach, even though it would be an overly rigid and dogmatic method for a team of experienced military minds.

G. TYPES OF OPERATIONS

Doctrine organizes operations into offensive, defensive, and stability categories. Our focus is on deliberate attack operations, which fall under the offensive category. To distinguish the attack from other types of offensive operations, we provide a brief description of each type here. The suitability of a type depends on the level of knowledge about the defending force and its status.

- **Movement to Contact:** when the offensive unit does not have enough information about the enemy to plan an attack in detail, it conducts a movement to contact. The intent is to establish contact and gain enough information to make an attack possible.
- **Attack:** the offensive unit has enough information to plan coordinated fire and maneuver to defeat the defender. A deliberate attack is planned when enough time is available for detailed coordination. A hasty attack, which by necessity must be simple enough to quickly communicate and execute, is conducted when the attacker's advantage is time-sensitive. A hasty attack typically follows a movement to contact.

- Exploitation: the attacking force takes advantage of the fleeing enemy to inflict heavy casualties. A large proportion of enemy combat casualties occurs during this type of operation. The ability to plan for detailed actions during exploitation depends on the predictability of the fleeing enemy's response. In real world situations, this is often left to subordinate commanders' discretion on the scene. However, if escape options are limited, exploitation planning could make this type of operation more devastating to the defender.
- Pursuit: can immediately follow exploitation. If possible, friendly forces chase down, cut off, and scatter or destroy the routed enemy. This prevents the enemy any chance to consolidate and prepare a coordinated response, and ideally wipes out enemy units entirely through casualties, capture, or desertion so they do not have to be fought again. Pursuit occurs over a wider geographical expanse than exploitation, so it is even more difficult to plan in detail beforehand. By its nature, it must react to the actions of the fleeing enemy while they remain disorganized.

H. PHASES OF ATTACK OPERATIONS

A common military technique for the management of time is to organize a plan into discrete, sequential phases (or stages, steps, etc.), where each phase has a unique focus. A plan can be phased however the commander sees fit, but most operations use a rather standardized sequence. An attack at the tactical level usually involves the following phases. This list uses a mix of Army and Marine Corps terminology; both services describe very similar phasing schemes in their doctrine. Standard phases could be used as the basis of an automated planning procedure; labeling the sequential steps of a plan in these terms would help with a user's understanding of a generated plan.

- (1) Preparation: reconnaissance elements determine or confirm enemy positions, capabilities, and intent. The unit conducts final rehearsals. Shaping fires commence.
- (2) Movement: shaping fires continue while the unit's elements move to the line of departure (LD), a tactical control measure that indicates the start of the attack once crossed. The LD is drawn so that friendly elements are not exposed to enemy observation or fire before crossing it, to include the routes to the LD. This allows some elements to wait short of the LD, if needed, to ensure subsequent maneuvers are properly coordinated.
- (3) Approach: friendly elements maneuver toward their objectives. The focus of supporting fires changes from shaping to protecting attacking units. Suppression of known enemy positions and screening of friendly units

(such as by smoke) are two common techniques. Maneuver units move as quickly as possible to their assault positions, attack by fire positions, or support by fire positions, balancing aggressiveness and force preservation.

- (4) Actions on the Objective: units take decisive action to secure their objectives. Attack by fire and support by fire units commence firing; assault units move from their assault positions and fight through their objectives, displacing the enemy. Supporting fires are ceased or shifted to avoid fratricide as friendly units approach.
- (5) Consolidation: friendly forces regroup on the objective, take account of their status, tend to wounded, redistribute ammunition, and prepare for a possible enemy counterattack. Commanders and planners begin preparing for the next mission.
- (6) Exploitation and Pursuit: an attack plan will often include exploitation and pursuit to attempt a seamless transition to those types of operations, assuming success of the attack. The level of specificity in this part of the plan depends on the attacker's ability to predict enemy actions once their positions become untenable.

When planning is promulgated down through multiple echelons, standard practice is to retain the phasing scheme of the higher echelons and only subdivide phases when needed for the lower-echelon command's concept of operations. This strict approach is not very useful when many echelons are involved because the time scales become much smaller as one proceeds down the chain of command. Additionally, large commands cannot transition from one phase to the next in an instant; inevitably, some subordinates end up performing actions in the next phase while others are still involved in the previous phase. If the prioritization of certain assets or command and support relationships change by phase, then an instantaneous transition may, in fact, be necessary to avoid ambiguity. The progress of the main effort is usually the best indicator of when the phase should officially transition.

I. MOVEMENT AND FORMATIONS

Combat troops work in relatively close proximity to one another to provide mutual support, mass fires on the enemy, and avoid being destroyed in detail. The same is true for the vehicles of a mounted unit. The geometric organization of a unit's members is called its *formation*. Individuals move in relation to the unit leader to maintain the shape

of the formation. Some units move quite amorphously in certain situations, such as when crossing an oddly shaped danger area or moving through contested urban terrain. For the sake of terminology, we include this type of movement under the umbrella term *formation*. In this sense, every unit in a combat situation is in some kind of formation unless it has been scattered—in which case it is no longer a unit.

Commands may organize their units into one or more formations as well. As in a formation of individuals, each unit in a formation moves with respect to its *base unit*, which is designated by the commander of the formation. A large formation of units may benefit from faster movement (in column) or more focused firepower (on line), but units in formation have less autonomy to take advantage of local terrain or react to enemy action. An OPORD often specifies large formation movements in the Movement phase, transitioning to individual unit movement as friendly forces proceed through the Approach and Actions on the Objective phases. Some tactics require units to move in separate formations. For example, a flanking attack or envelopment requires the maneuvering element to separate from the fixing element.

When area-fired weapons such as artillery are a threat, commanders keep their formations spread out to avoid being decimated by a single volley. Medium and heavy direct fire weapons are more effective when targets are lined up parallel to the direction of fire, so unit leaders choose formations that limit this possibility. Finally, troops and vehicles must generally be lined up perpendicular to the direction of the enemy—otherwise, they will block each other's lines of fire. These considerations lead to particular formation shapes optimized for different situations. For example, when multiple units assault a single objective area, an on-line formation is the usual choice to maximize firepower to the front while minimizing the chance of fratricide.

To prevent units from crossing over each other, planners use tactical control measures (Figure 77) to restrict commands to different regions of the AO. The most common approach to this is the assignment of mutually exclusive AOs (that is, a partition of the senior command's AO) to each individually moving command—for example, the lines marked “XX” in Figure 77. This allows each command to choose its own detailed routes within the AO boundaries. Phase lines, which can stretch across multiple AOs, can

also be used to control or mark the advance of multiple units simultaneously. PL LYNN and PL LILLY in the same figure provide examples of phase lines that will be crossed by units traveling along AXIS EDWARDS.

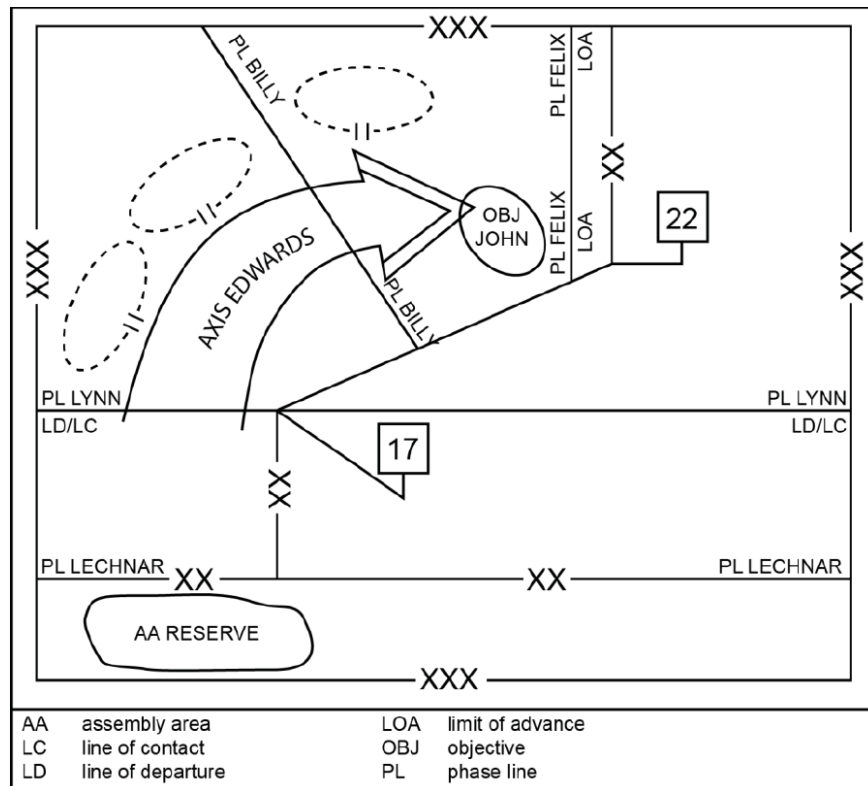


Figure 77. Tactical Control Measures. Source: DOD (2012a).

J. FORMS OF MANEUVER

Current military doctrine uses six characteristic approaches to conducting an attack, called the *forms of maneuver*. Each is defined with respect to the application of friendly forces against the enemy force's front, flanks, and rear, and the intended enemy response. We briefly describe the forms of maneuver here; the U.S. Army's field manual for offense and defense provides useful diagrams and lists typical control measures for each form (DOD 2013, 1–2 to 1–22). Several forms of maneuver share common attributes, and often a real world scheme of maneuver falls somewhere on the spectrum between two or more of them.

- Frontal Attack (Figure 78): the attacking force attempts to overwhelm the defending force rapidly by a simple, direct maneuver against its front, where the preponderance of the defender's combat power is expected to be oriented. It prioritizes speed and simplicity. Depending on constraints, it may be the only option.

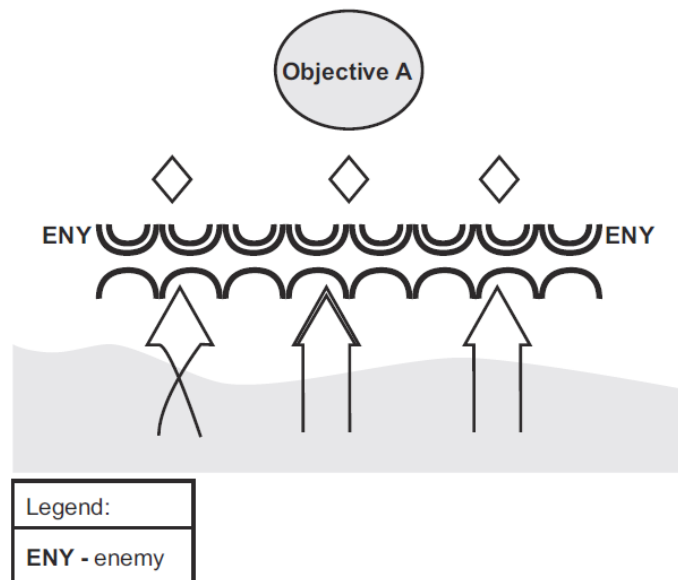


Figure 78. Frontal Attack. Source: DOD (2014a).

- Flanking Attack (Figure 79): the attacking force strikes at the right or left flank of the defender's formation, where his combat power is expected to be weaker, with the intent of defeating the defender in place. This is usually supported by a fixing maneuver at the defender's front. A flanking attack prioritizes relative combat power and enemy immobility.

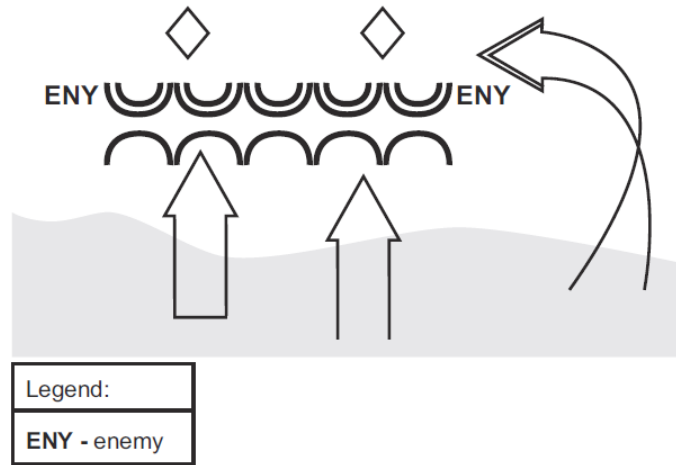


Figure 79. Flanking Attack. Source: DOD (2014a).

- Envelopment: the attacking force bypasses the preponderance of the enemy formation to defeat valuable elements in the rear area, resulting in the defeat of the entire enemy force. Similar to a flank attack, a fixing maneuver at the front often contributes to protecting the main effort from most of the defender's combat power. An envelopment intends to defeat the enemy in its current location. It prioritizes relative combat power, positional advantage, and prevention of enemy maneuver.
- Turning Movement: the attacker strikes at a valuable element or position of the defender with the intent of causing the defender to displace some of his forces as a reaction. This intent distinguishes a turning movement from the other forms of maneuver. Turning movements prioritize positional advantage and relative combat power.
- Infiltration: the attacker breaks up into small units and bypasses the enemy front and gain a better position from which to attack. Infiltration prioritizes stealth, positional advantage, and relative combat power.
- Penetration: the attacker breaks through the enemy front by the application of combat power in a relatively small area and then takes advantage of its position in the defender's rear to defeat the entire force. It prioritizes relative combat power, speed, and positional advantage. It favors simplicity more than an envelopment or infiltration because the attacking forces are less geographically separated.

A careful reading of doctrine for the various echelons reveals that certain forms of maneuver are more appropriate for units of a certain size. At the small unit level, forms of maneuver tend to be replaced by tactical tasks. The canonical example of this is a squad

tasked to establish a support by fire position for a platoon flanking attack. Similarly, a brigade planning a flank attack would likely task one of its battalions with conducting an attack to fix the enemy in place as a supporting effort. That battalion would likely describe its scheme of maneuver as a frontal attack, but its constituent small echelons would use attack by fire tasks. This example also illustrates how each unit can employ a different form of maneuver within the framework of a larger plan.

The following (non-doctrinal) table represents some of the tactical tasks and forms of maneuver that are most likely to be used by different echelons. Those marked X are commonly considered; those marked (X) are possible but less likely, and those with no mark are rarely considered. These are not absolutes, but should be a useful guide for prioritizing modeling efforts.

Table 11. Tactical Tasks and Forms of Maneuver by Echelon

		Tactical Tasks			Forms of Offensive Maneuver						
		Support By Fire	Attack By Fire	Breach	Frontal Attack	Flank Attack	Envelopment		Turning Movement	Infiltration	Penetration
							Single	Double			
Tactical Echelon	Fire team	X	X	X	X						
	Squad	X	X	X	X	(X)					
	Platoon	X	X	X	X	X				X	
	Company	(X)	(X)	(X)	X	X	(X)			X	
	Battalion				X	X	X		(X)	X	(X)
	Brigade				X	X	X	(X)	(X)	X	X
	Division				X	X	X	X	X		X

From recommendations in echelon-specific doctrinal publications in the U.S. Army's ADRP and FM 3-series and USMC's MCWP 3-series.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. TESTING SCENARIO DESCRIPTIONS

A. INTRODUCTION

We present here overviews of each scenario group. Our focus is on the tactics, for two reasons: to argue for the relevance of each scenario to realistic combat operations, and to provide a basis for subjective analysis of the fire support planner's output. We illustrate each scenario group with screen shots, and we provide a brief textual explanation of the offensive and defensive tactics (unit placements and Route designs) chosen as input for the fire support planner. All cardinal directions (north, east, etc.) refer to the edges of the figure (top, right, etc.).

We provide some additional explanation of icons, symbols, and tactics in the description of Scenario Group 1.

B. SCENARIO GROUP 1: PLATOON VERSUS SQUAD, MAP A

Scenario Group 1 is the smallest and simplest. It consists of a platoon attack on a squad position on Map A. Figure 80 shows the Enemy Force's arrangement, which consists of three fire team battle positions. The observation post is the easternmost-pictured group of entities. The large red cubes are highlighting boxes that help the user identify entity locations from a wide zoom level; the entities themselves are smaller capsule-shaped objects inside each cube. The small red dots above the entities are unit icons. Highlighting boxes and unit icons are for display only; they have no physical meaning during a replication.

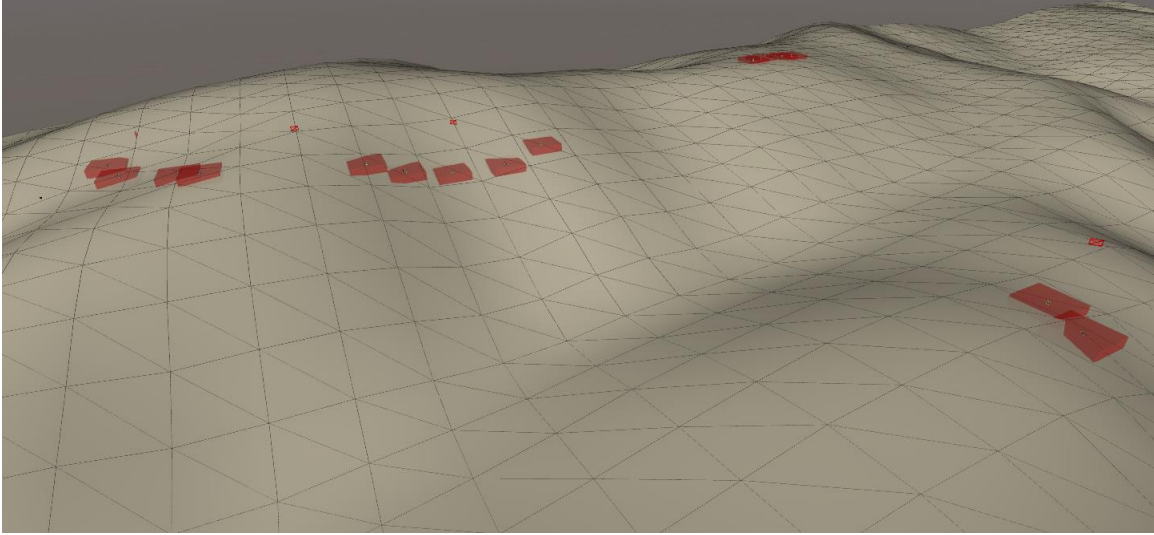


Figure 80. Scenario Group 1 Objective Area

Figure 81 depicts the portion of the map threatened by the Enemy Force. The occluded regions (areas with no red triangles) near the defensive positions are candidates for assault positions, where the Friendly Force should be able to transition to online formations prior to beginning their assault movement.

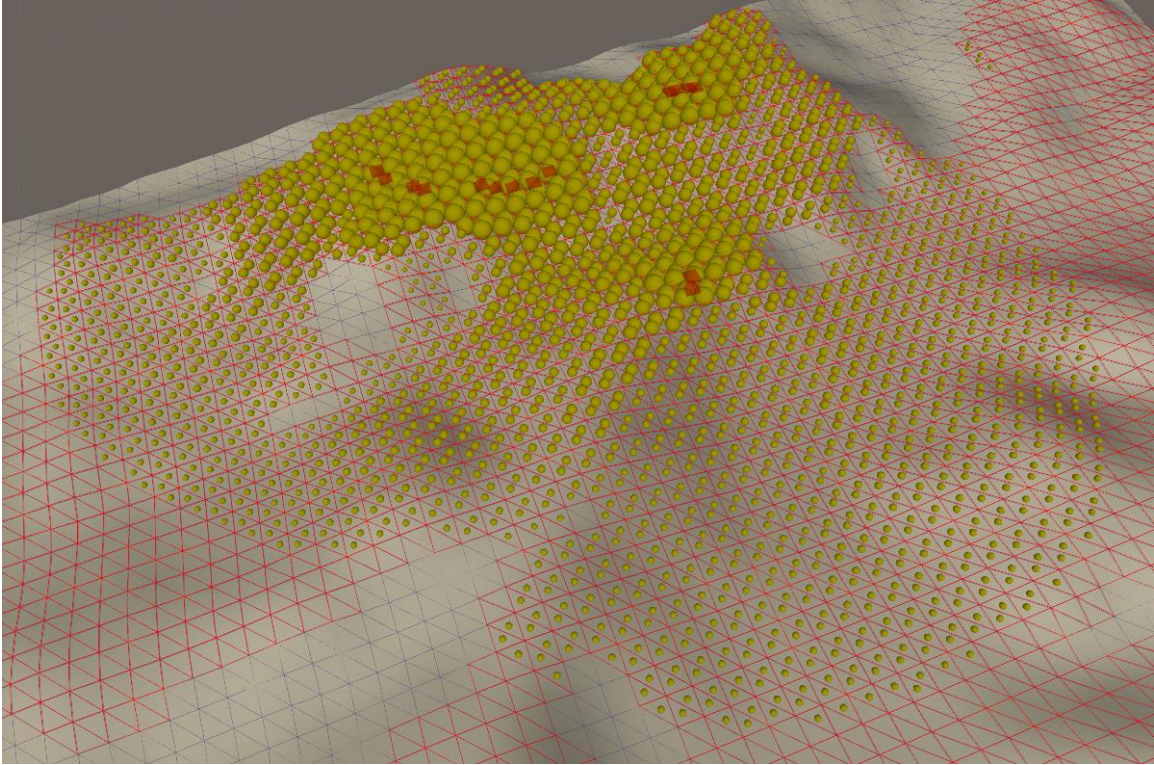


Figure 81. Scenario Group 1 Node Penalty Visualization

The Friendly Force starting positions and maneuver plan are shown in Figure 82. The blue arrows starting from the blue highlighting boxes depict squad movements to assault positions. The shortest two of these are the machine gun squads' movement to their manually assigned firing positions, which are only used for the manual planning type scenario. The east-to-west arrows show the directions of assault for the rifle squads; the one closer to the camera is actually a pair of arrows from a single assault position to two different Enemy Force units. The end of each assault arrow is centered on an AssaultObjective object whose radius determines which entities the assaulting squad should engage with close-range fires. We manually place each AssaultObjective such that its radius captures an Enemy Force fire team and any nearby single-entity leaders, such as a squad leader or platoon commander.

This selection of assault positions and objectives is meant to depict a simple flank attack.

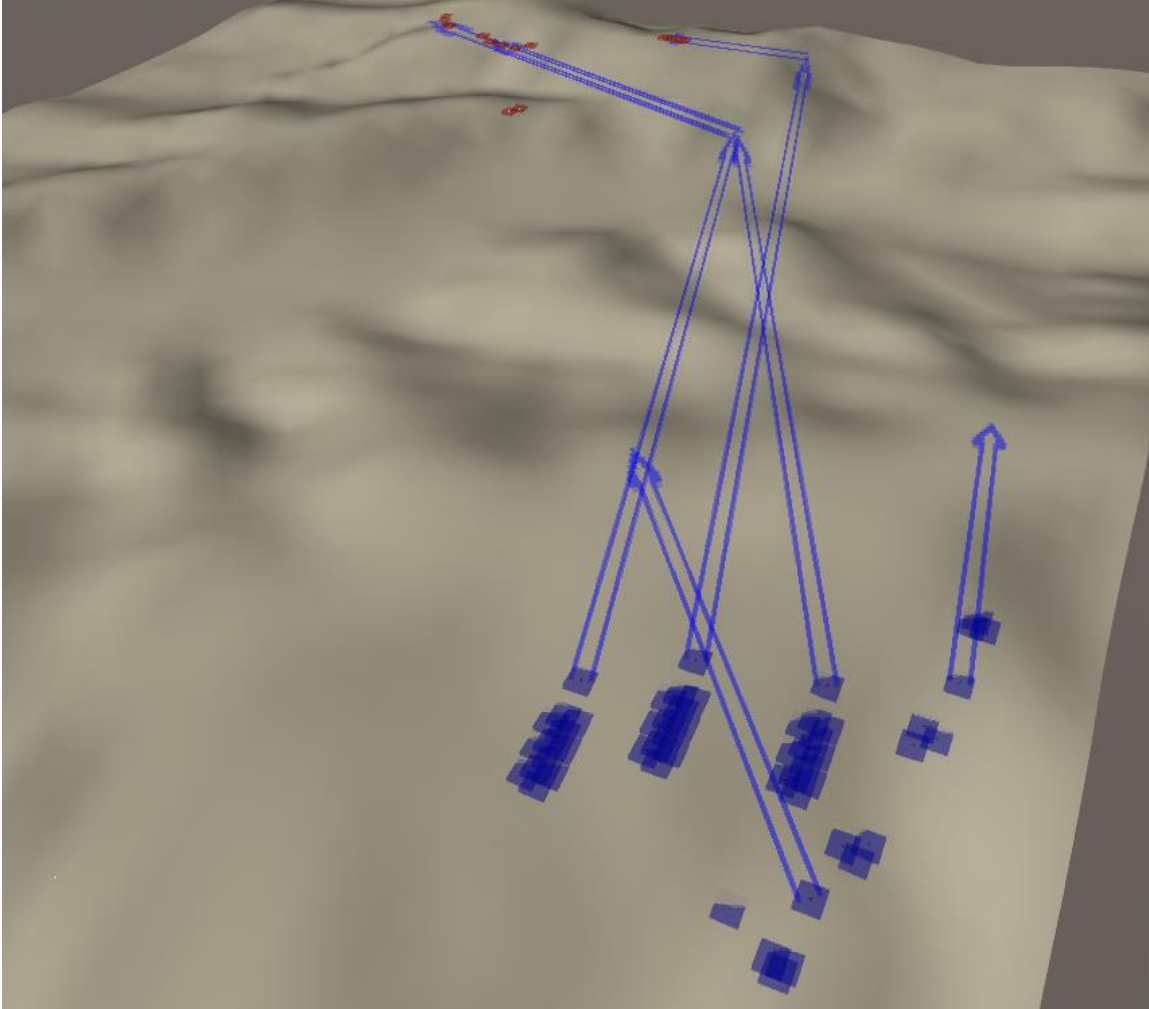


Figure 82. Scenario Group 1 Maneuver Plan

Figure 83 shows the alternate machine gun squad starting positions for the close-quarters scenario. As with all close-quarters scenarios, the maneuver units' starting positions and tactics have not been changed. Note that one of the machine gun squads has a safe path away from the objective area, but the other squad does not.

The highlighting boxes tend to alter a human observer's perception of distance because they make the entities appear much larger than they actually are. The left and right machine gun squads in Figure 30 are approximately 200m and 300m, respectively, from the closest enemy entity. This is still unreasonably close to the defender positions for a fire support unit, but the close-quarters scenario is contrived to test the robustness of

the planning algorithm. If desired, one can invent a narrative to explain this arrangement of forces—perhaps the fire support units happened upon a previously undetected enemy battle position. The Enemy Force, if it had any movement capability and tactical sense, would attempt to move into weapons range of the machine gun squads, possibly resulting in the situation of Figure 38 at some intermediate point in time.

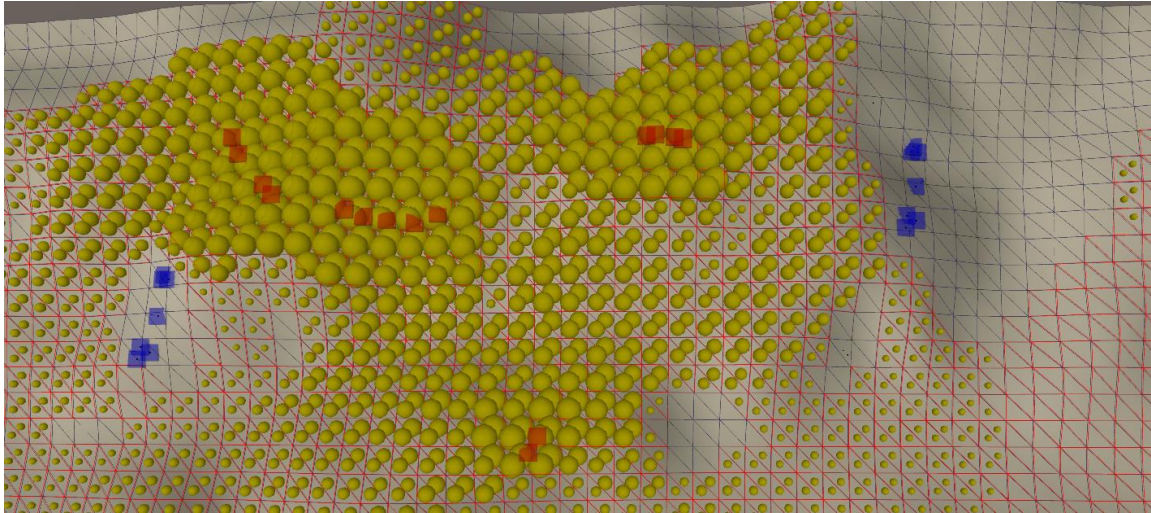


Figure 83. Scenario Group 1 Close-quarters Starting Positions

C. SCENARIO GROUP 2: PLATOON VERSUS SQUAD, MAP B

Scenario Group 2 uses the same force pairing as Group 1 on the medium-sized Map B. The defensive position includes one fire team oriented northwest and two fire teams oriented southwest (Figure 84). Due to the larger area for potential Friendly Force maneuver, we have chosen an observation post position farther away from the objective area than that of scenario group 1.

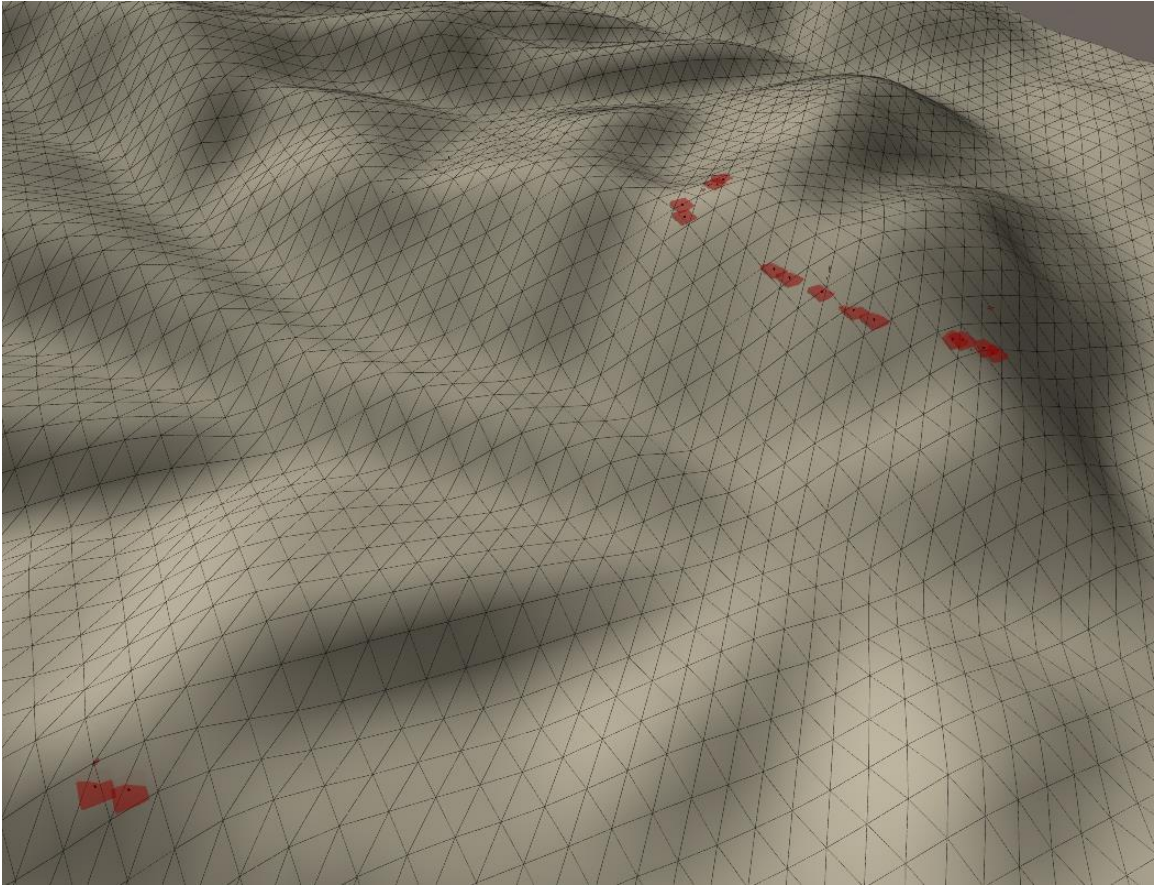


Figure 84. Scenario Group 2 Objective Area

The more distant observation post allows the Enemy Force to threaten a larger area of the map (Figure 85). A large area behind the battle position is left unobserved, but we assume that the attackers do not have access to it.

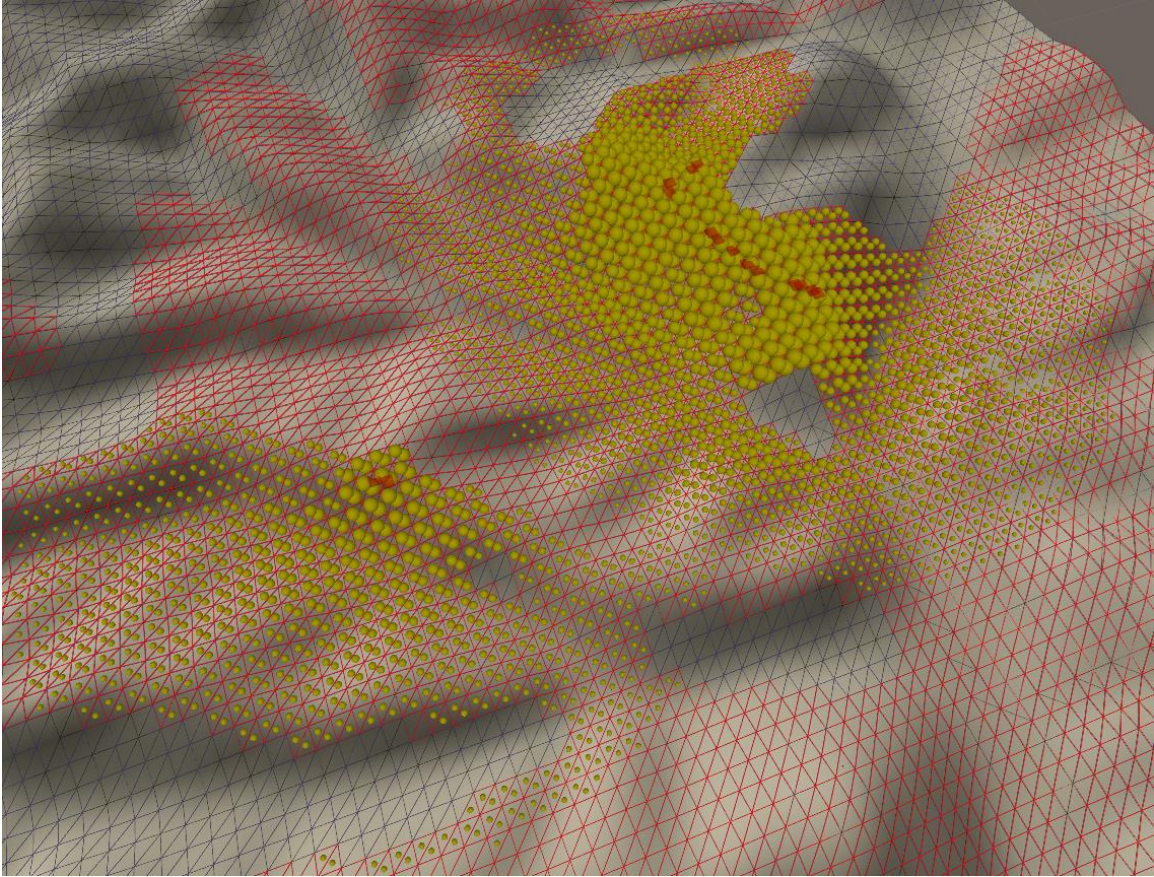


Figure 85. Scenario Group 2 Node Penalty Visualization

The tactics of the maneuver plan (Figure 86) amount to a left flank attack. The movement from the assembly area to the assault position (shown by the longest blue arrows) appears to be threatened by the Enemy Force observation post, but it turns out that there is a mostly safe path—not on a straight line—to transit between the observation post and the battle position (in Figure 85, the area between the observation post and main defensive position with no yellow markers). The arrow towards the bottom of the figure show the movement to the manual fire support positions, which are only used with the manual planning type.

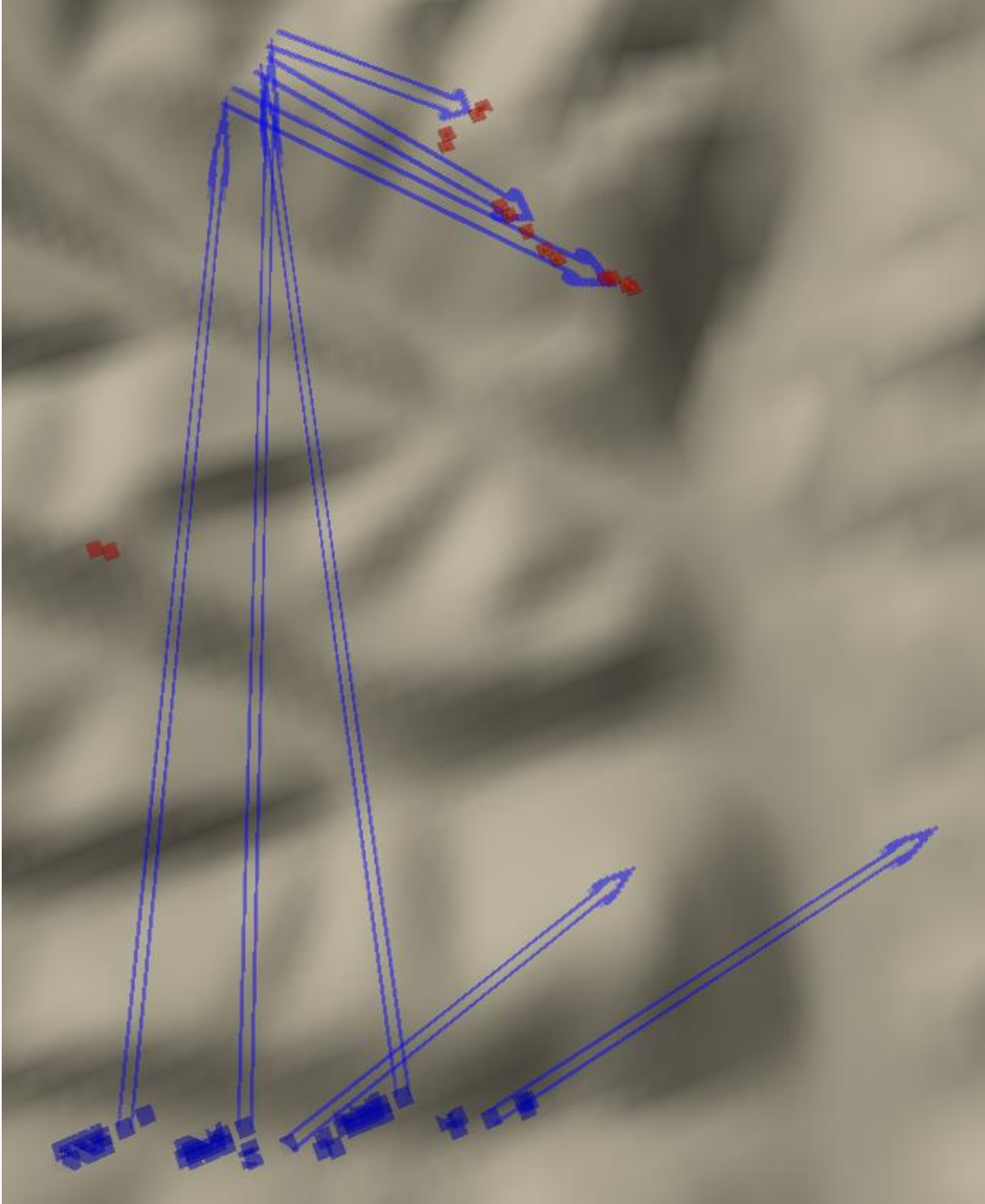


Figure 86. Scenario Group 2 Maneuver Plan

The close-quarters positions (Figure 87) do not use the open area behind the objective area due to the same assumption that caused that area to be uncovered: that the

Enemy Force's rear area is inaccessible. Nonzero-risk nodes surround both of the starting positions for machine gun squads, but the squad farther from the camera has a mostly safe path out of the Enemy Force's weapon range.

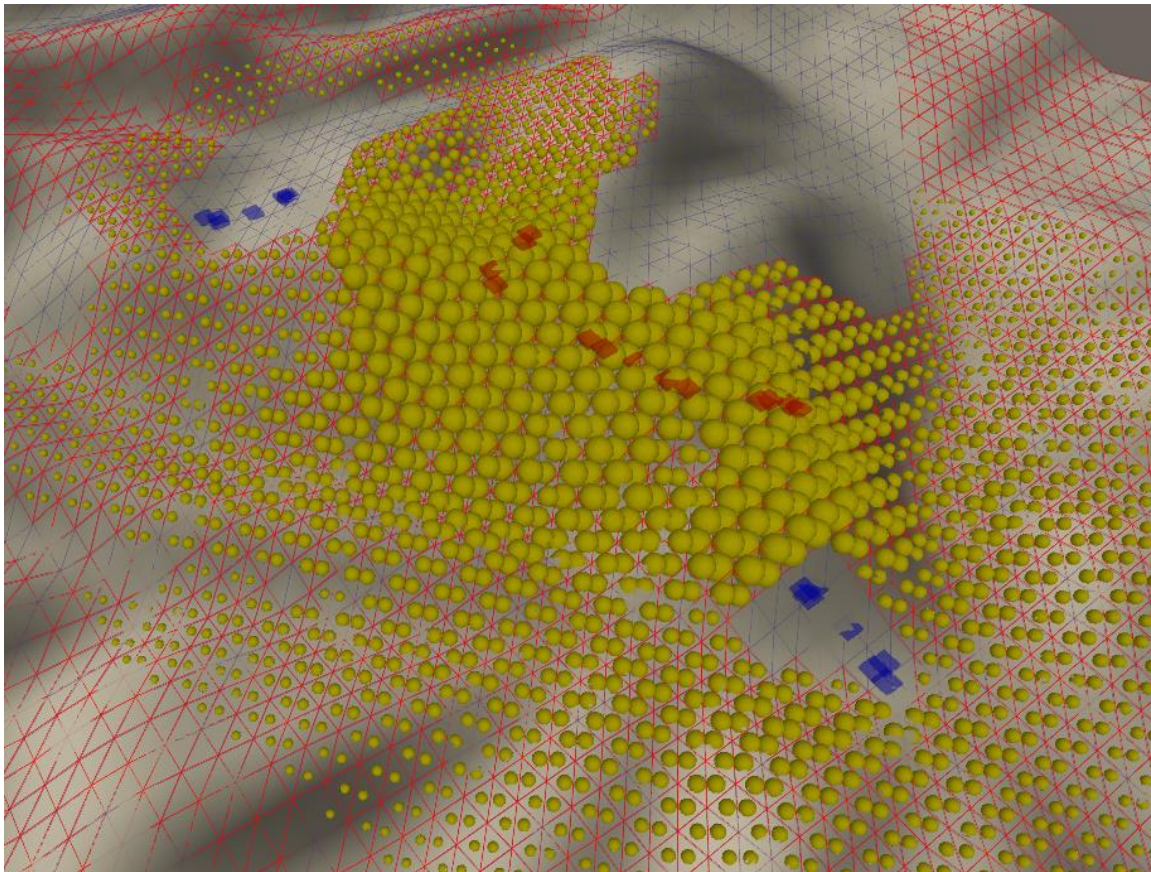


Figure 87. Scenario Group 2 Close-quarters Starting Positions

D. SCENARIO GROUP 3: COMPANY VERSUS PLATOON, MAP B

Although a company-sized attack could conceivably fit in a 2 km by 2 km area such as that of Map A, there would be little room for different maneuver options. We therefore restrict the company-level attacks to Maps B and C. The objective area for Scenario Group 3 lies atop the same geographical feature as that of Scenario Group 2, but it covers a larger portion of it due to the larger Enemy Force (Figure 88). The tactic of this battle position is a reverse-slope defense, meaning that the defenders are placed to fire on the attackers from the moment they crest the next adjacent hills until they reach the

objective area. This explains their placement approximately halfway between the top and base of the geographical features they occupy. The general layout of the battle position is on line, with a J-hook on the east side to rebuke a potential flanking attack. The observation post is placed on the next significant terrain feature south of the position.

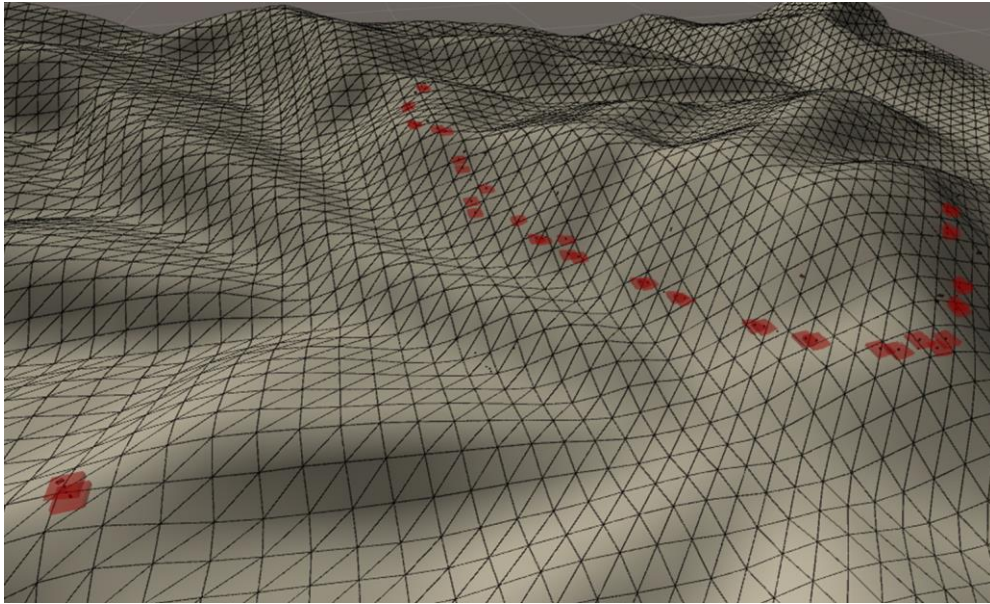


Figure 88. Scenario Group 3 Objective Area

Figure 89 illustrates how moving from a squad- to platoon-sized battle position significantly increases the complexity of the Enemy Force's terrain coverage—and lack of coverage. Several areas of dead ground appear due to the many small valleys of Map B. As always, most of the immediate area around the battle position is high-risk due to careful entity placement, but there is enough variety in the terrain relief to offer a few pieces of defilade for the attacker. Although the observation post has some significant dead ground to its south, it can cover most of the terrain at the edge of its weapons range even farther south.

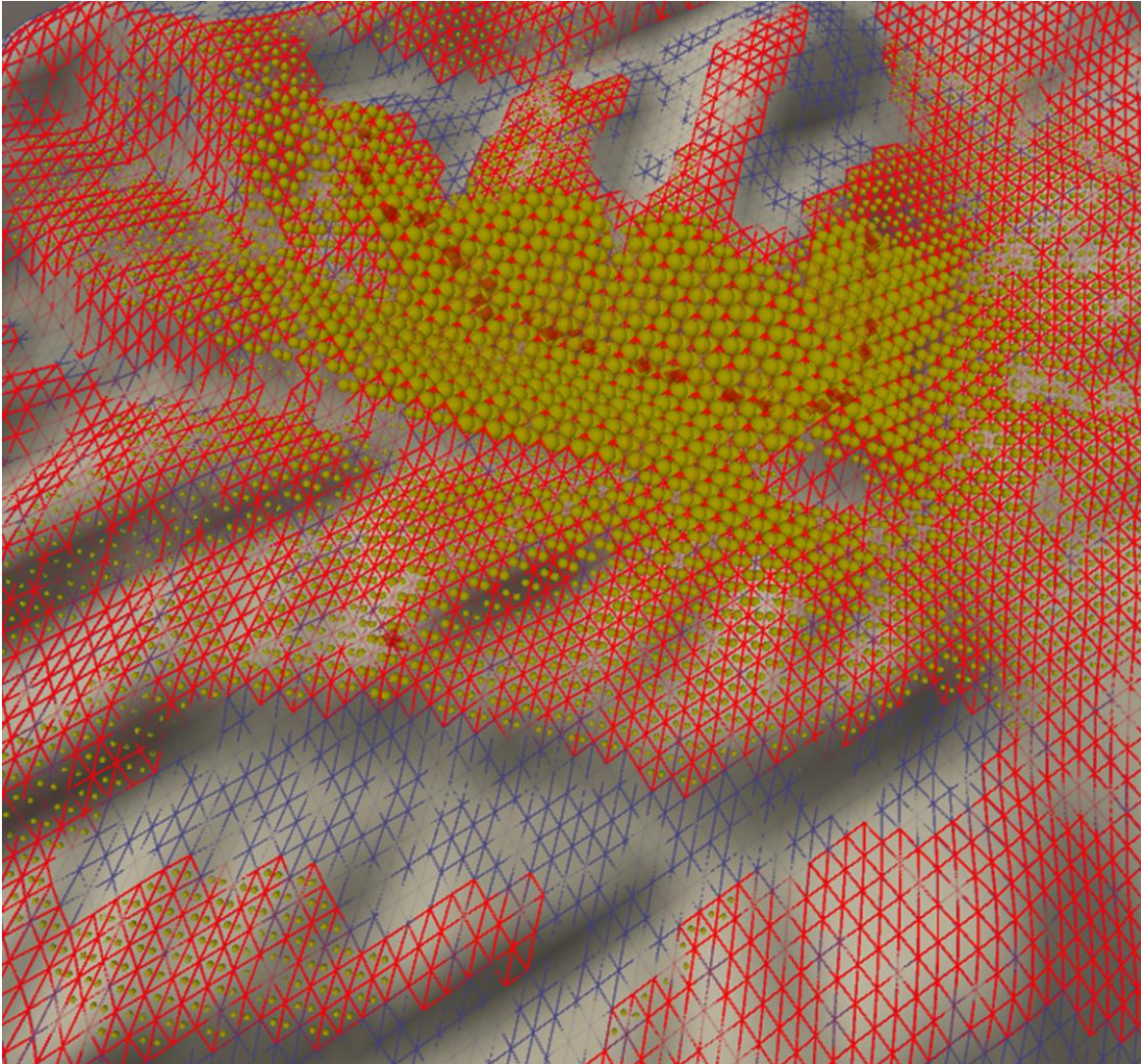


Figure 89. Scenario Group 3 Node Penalty Visualization

There are several possible tactics for attacking this objective area. The one we have selected might be described as a sequential frontal and right flank attack. The idea is to first overwhelm the eastern J-hook part of the battle position, then sweep across from east to west in a series of flanking attacks (Figure 90). Prior to the main attack, we have chosen to assault the observation post with one squad to prevent it from harassing the other units' assault positions. As always, the blue arrows that do not terminate on an Enemy Force position are the manual fire support positions.

This is the first scenario group to use platoon-level movements. Each of the three platoon attacks consists of a single platoon movement (shown as one blue arrow) to an assault position, followed by two or three squad assaults (shown as two or three blue arrows). Within each platoon attack, all of the squad assaults are sequential. This is important for fire support planning because it allows the possibility of each attacking squad to support subsequent squad assaults.

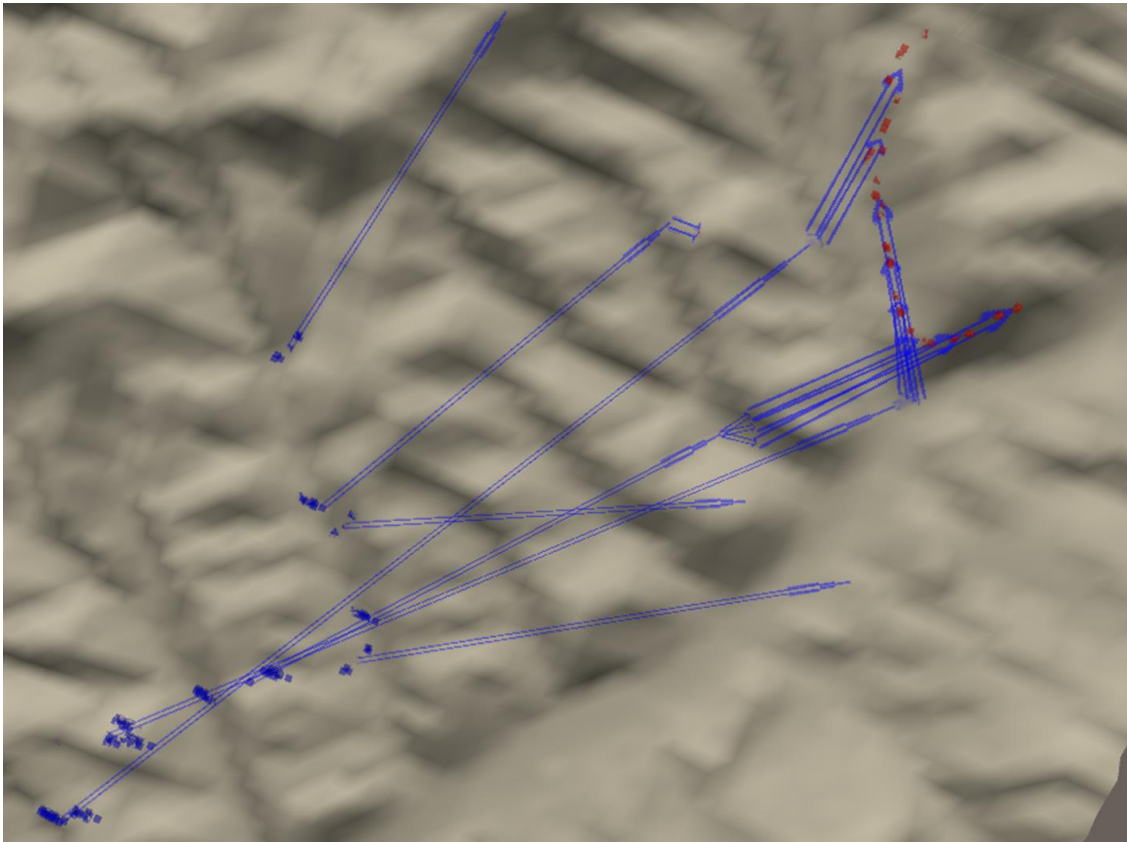


Figure 90. Scenario Group 3 Maneuver Plan

This scenario group offers several options for close-quarters starting positions. Our selections are shown in Figure 91. Of note, all of these positions offer safe paths outside of Enemy Force weapons range.

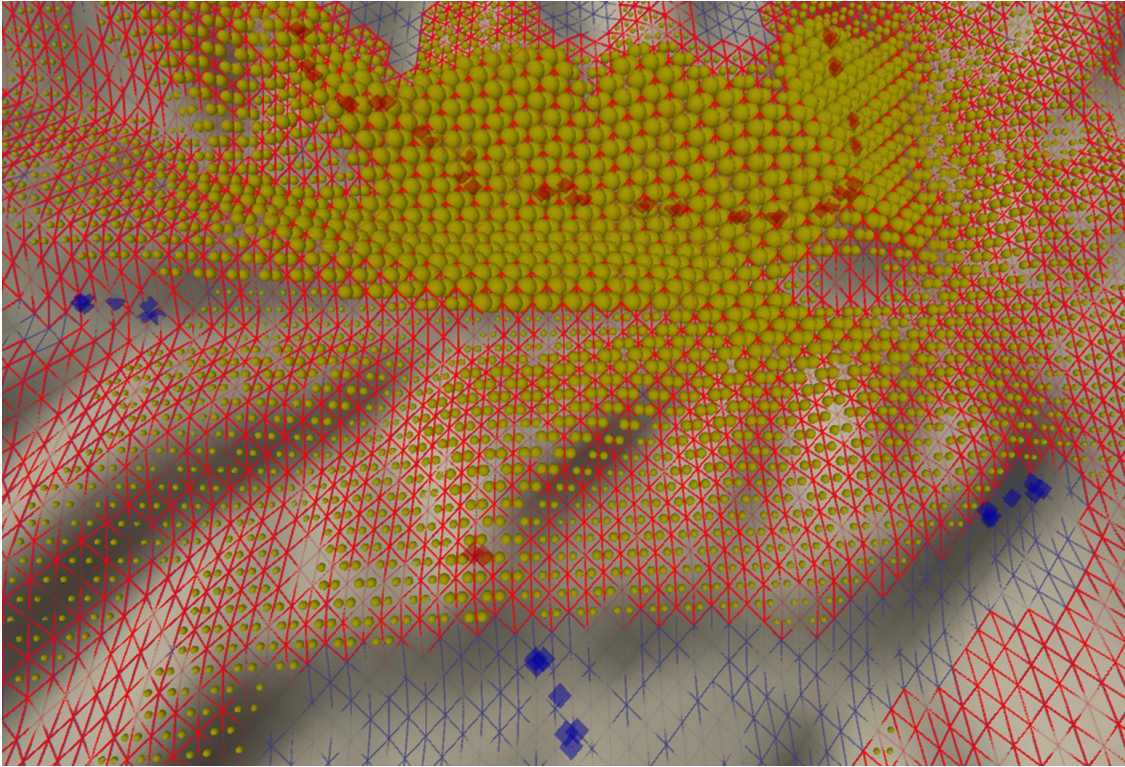


Figure 91. Scenario Group 3 Close-quarters Starting Positions

E. SCENARIO GROUP 4: PLATOON VERSUS SQUAD, MAP C

The final platoon versus squad scenario uses the very large Map C, but the small collection of units only covers a fraction of its total area. The defensive position is laid out quite simply, with the observation post positioned to cover dead ground on the east side (Figure 92).

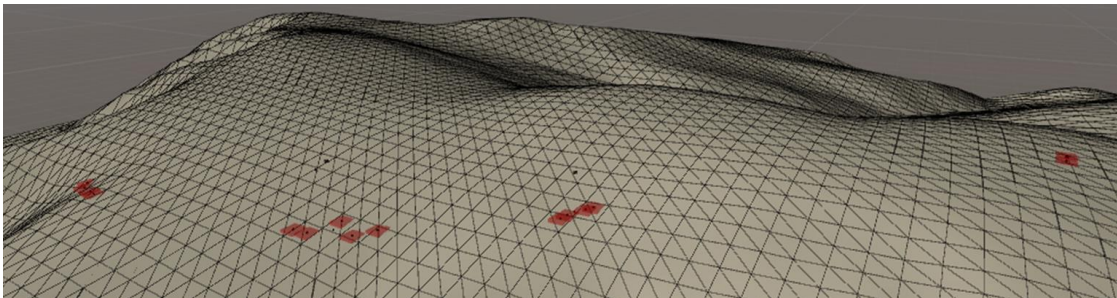


Figure 92. Scenario Group 4 Objective Area

Although the terrain of Map C has significant mountains and valleys, the immediate surroundings of the objective area for Scenario Group 4 is relatively flat. The only protected approach path leads almost straight up the middle of the Enemy Force's frontage (Figure 93).

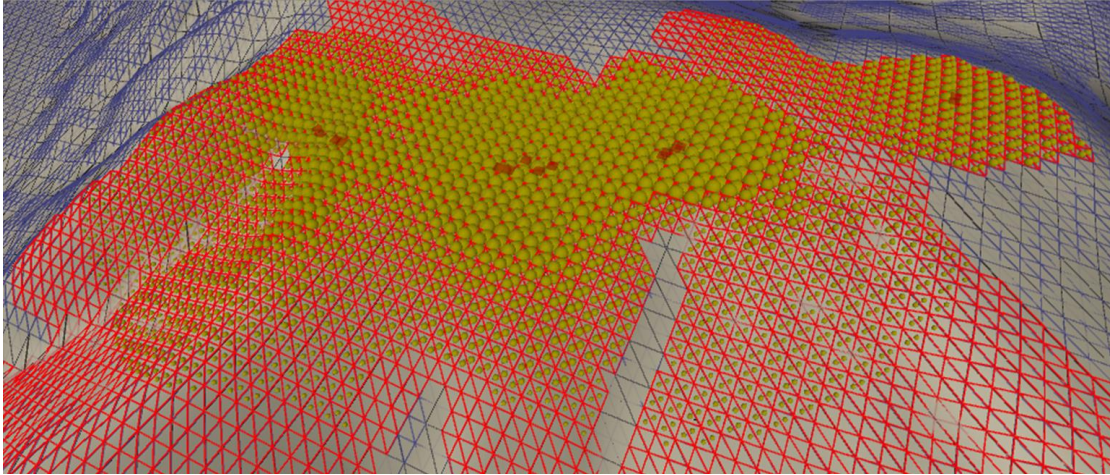


Figure 93. Scenario Group 4 Node Penalty Visualization

The form of maneuver for Scenario Group 4 is a frontal attack (Figure 94), which contrasts with the flanking attacks of Scenario Groups 1 and 2. The variety in the tactics is meant to test the fire support planner against different types of maneuver plans. One of the challenging aspects of a frontal attack like this one is that all of the assaults occur near-simultaneously, so the fire support units cannot support all of them. The arrows with the southernmost endpoints show the movement to the manual fire support positions.

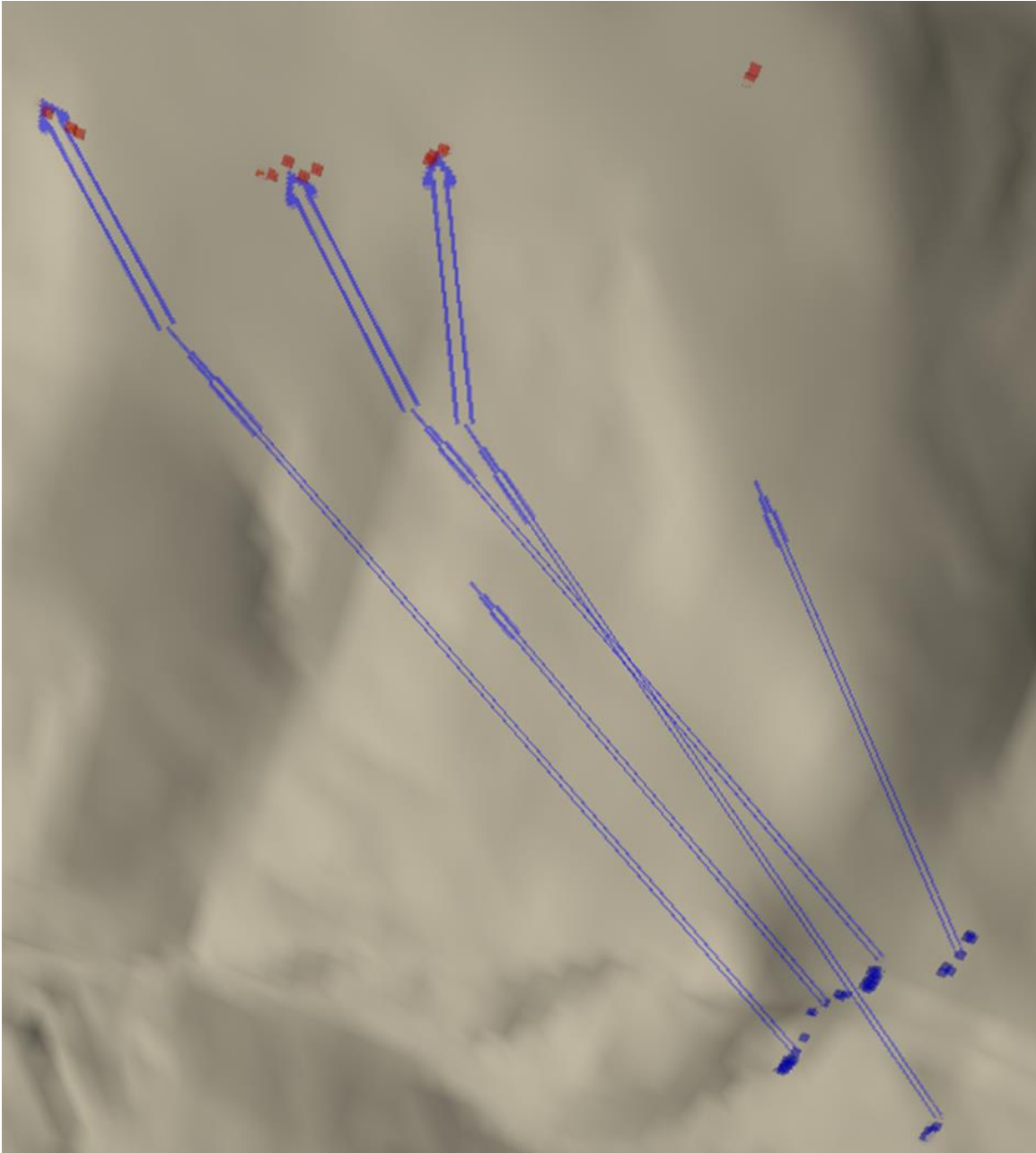


Figure 94. Scenario Group 4 Maneuver Plan

For close-quarters positions, the only options—besides the out-of-bounds area behind the objective area—are a portion of the aforementioned central approach path and a position in defilade from the observation post (Figure 95).

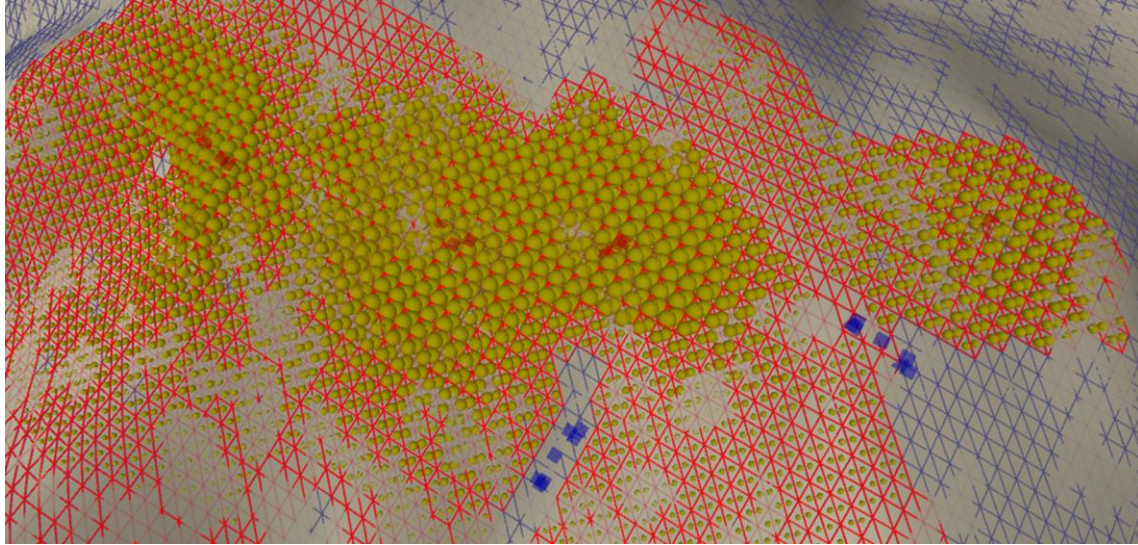


Figure 95. Scenario Group 4 Close-quarters Starting Positions

F. SCENARIO GROUP 5: COMPANY VERSUS PLATOON, MAP C

Since Map C covers a large area (8 km by 8 km), we have chosen a relatively widely dispersed defensive mission for the Enemy Force (Figure 96). The narrative idea of this situation is that a main supply route flows through the low ground on the western edge of the image, and the Enemy Force is positioned to threaten it. The Friendly Force must clear Enemy Force units to eliminate the threat (allowing notional supplies to flow unimpeded, etc.). The Enemy Force units east of the main supply route are meant to protect the main force from flanking attacks. In this defensive plan, the observation post is placed on the high ground inside the battle position. One fire team, found in the center of the image, is placed to protect against an attack up the draw (low ground) leading behind the main force.

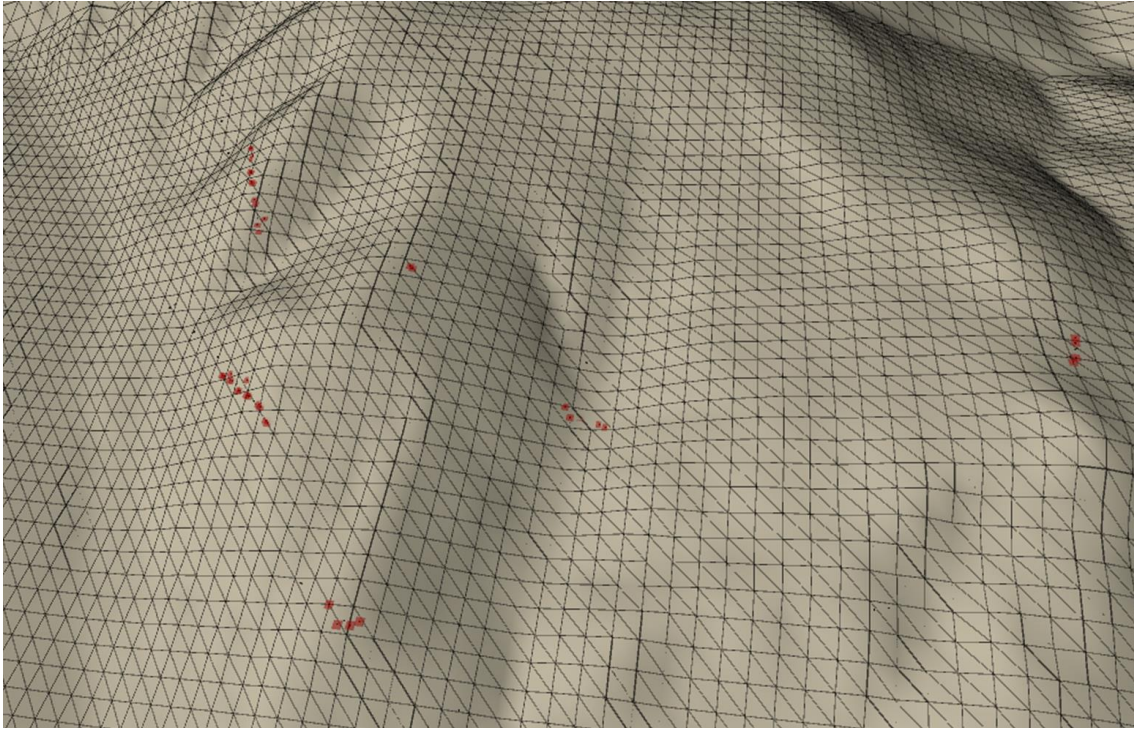


Figure 96. Scenario Group 5 Objective Area

Although Scenario Group 5 has the same force pairing as Scenario Group 4, the broader geographical coverage produces a more interesting—though perhaps less-effective—threatened area (Figure 97). There are no completely protected approaches to defender positions, but there are few interlocking fields of fire above the squad level.

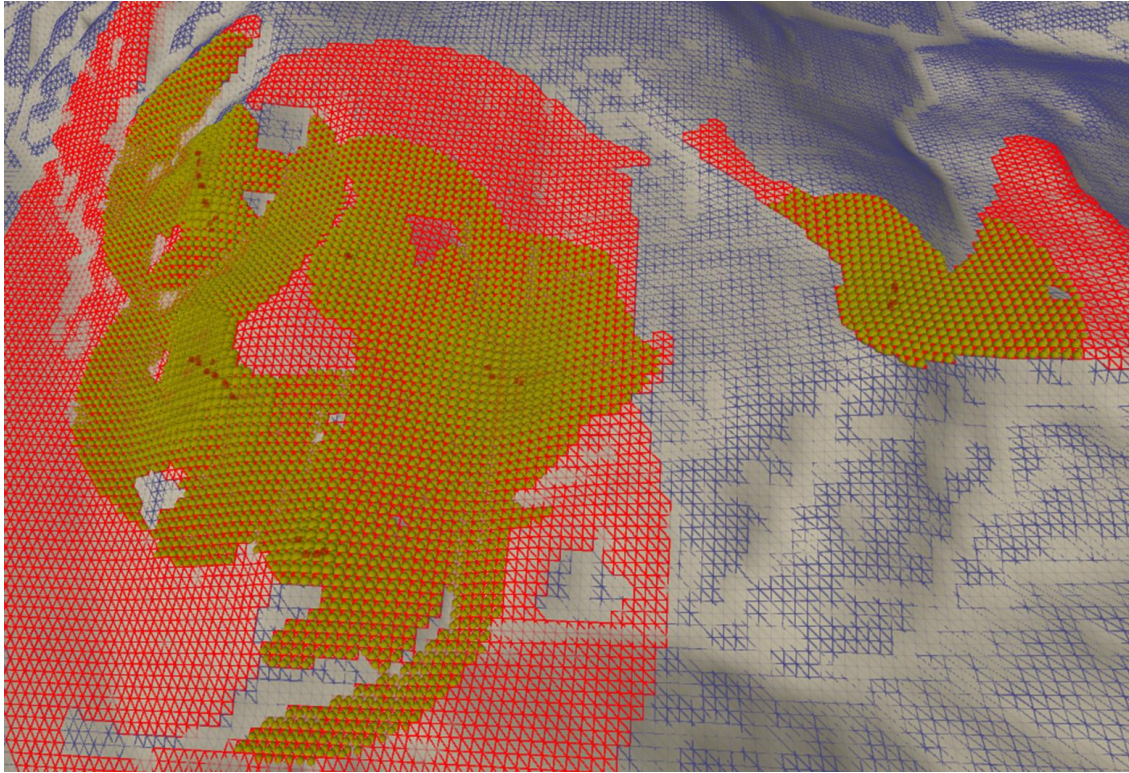


Figure 97. Scenario Group 5 Node Penalty Visualization

The maneuver plan for Scenario Group 5 (Figure 98) is best described as a right flank attack at the company level. However, the maneuver plan plays out more like three separate platoon attacks than a coordinated company attack. This differs from the more coordinated appearance of Scenario Group 4, probably due to greater separation of the Enemy Force's squad positions in Scenario Group 5.

The two westernmost manual fire support positions are located in the base of the valley that is notionally threatened by the Enemy Force positions. These would not be very realistic positions in the real world due to the ballistics of firing up a slope. But since these details are not modeled in WXXI, the valley positions end up offering more targeting options for the attackers. The fire support planner is also able to use positions on the valley floor, so we do not consider these manual positions to offer an “unfair” advantage.

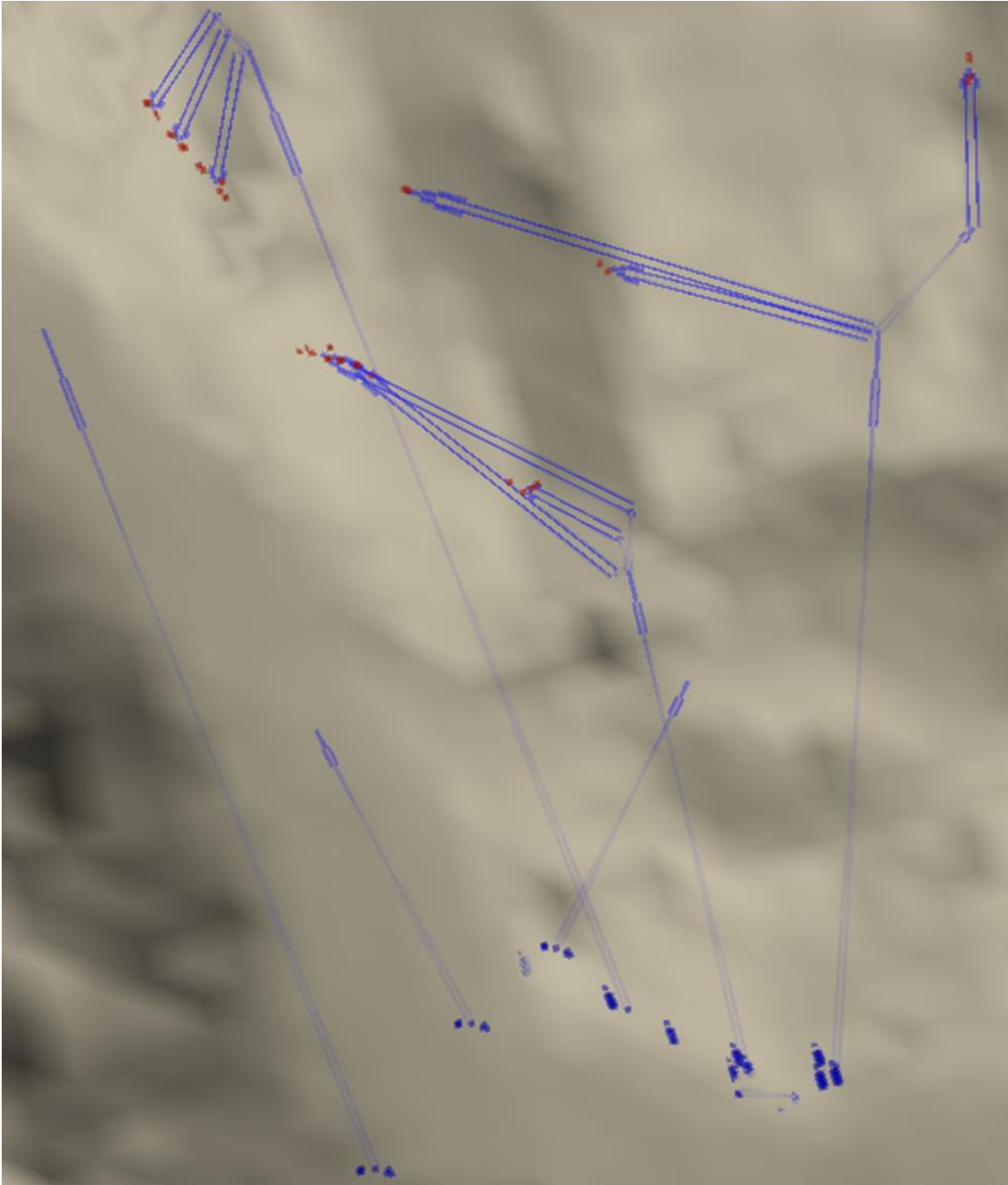


Figure 98. Scenario Group 5 Maneuver Plan

There are many occluded areas to choose from here for close-quarters starting positions (Figure 99). We have selected one near the easternmost Enemy Force position.

The westernmost starting position has no safe paths to a useful firing position. This is meant to stress the fire support planner.

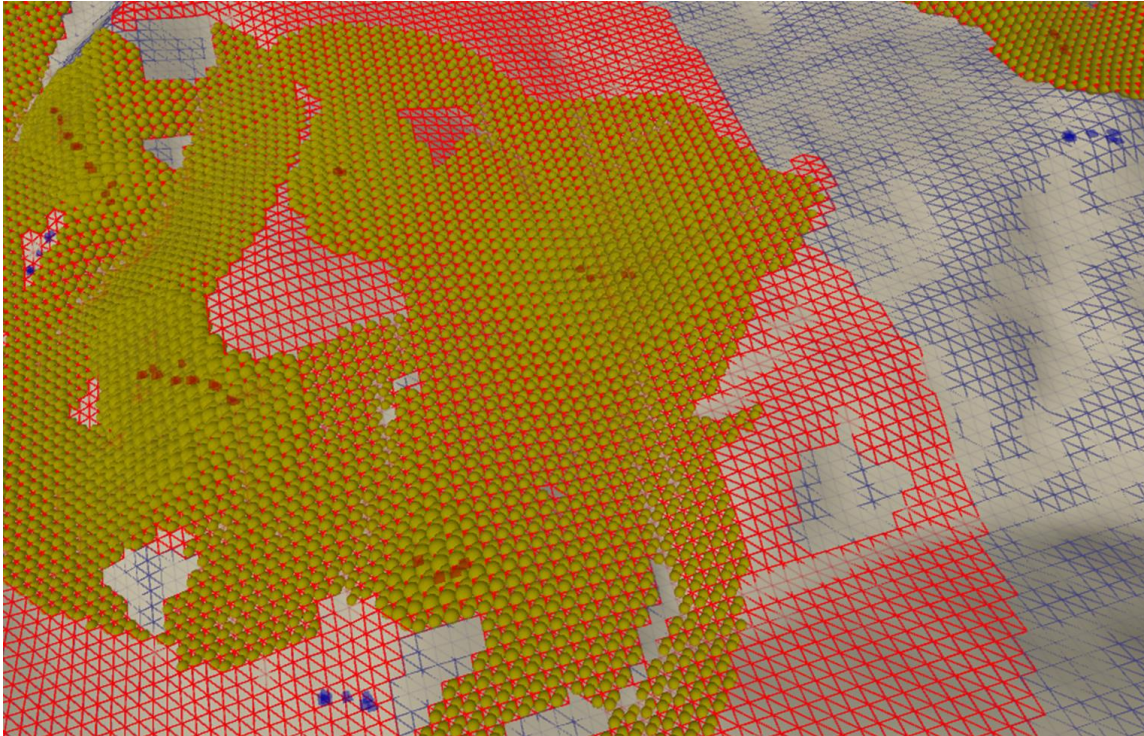


Figure 99. Scenario Group 5 Close-quarters Starting Positions

G. SCENARIO GROUP 6: BATTALION VERSUS COMPANY, MAP C

The final scenario group features an entire infantry company defending the same general objective area as Scenario Group 5 (Figure 100). The narrative explanation is the same: threatening the supply route through the valley. In this case, the Enemy Force has much more combat power to protect its left flank (on the east side). The main challenge for the attacker is that almost the entire defensive frontage has dangerous overlapping fields of fire. The Enemy Force's observation post is located on the protruding terrain feature between the right flank (west) and the front (south). The small unit in the center of the battle position is a normal fire team, but is meant to represent the company headquarters.

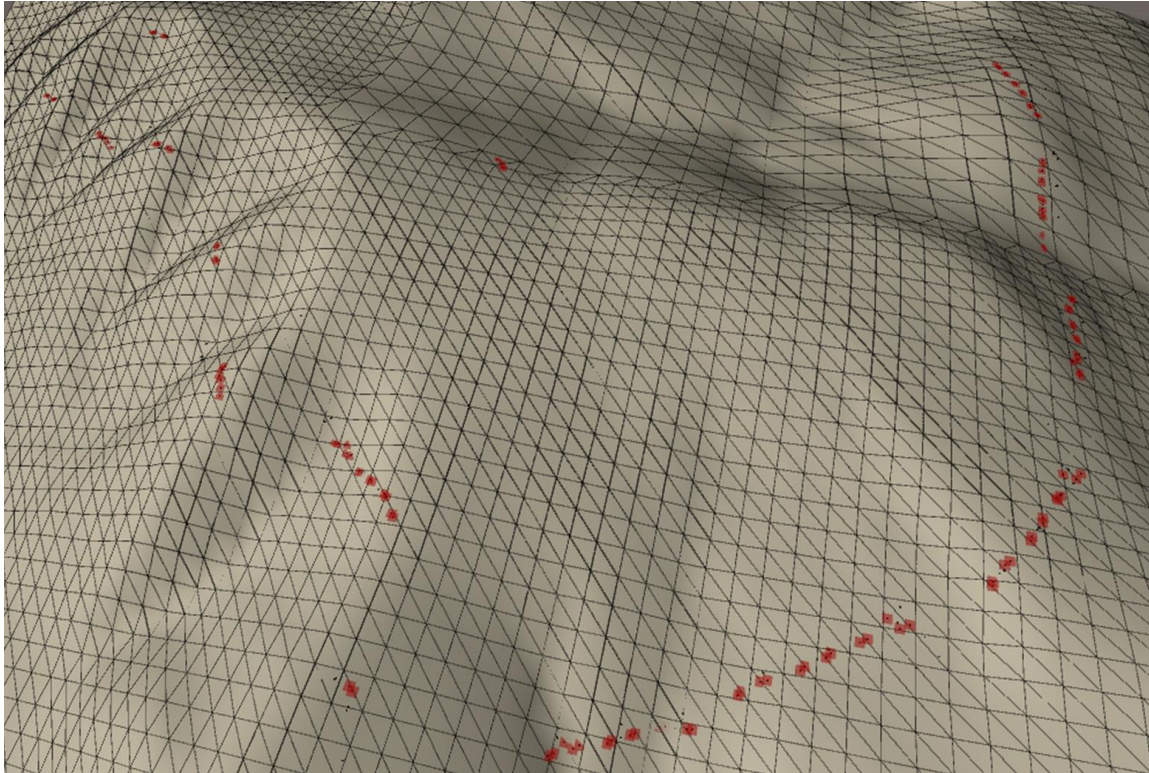


Figure 100. Scenario Group 6 Objective Area

The node penalty visualization (Figure 101) reveals a significant difference in the terrain on the western and eastern flanks of the Enemy Force. The western flank, adjacent to the valley, has sharp changes in elevation and slope, making it difficult to cover with observation and fires. However, the generally steep slopes facing the valley are not supportive of an attack.³⁵ The eastern flank is more level, making it dangerous for a frontal attack even though the terrain appears to support it.

³⁵ The movement model of WXXI is not mature enough to slow down entities ascending a steep slope to the same degree we would expect in the real world. However, the pathfinding module does not allow node traversals with vertical slope greater than 30 degrees.

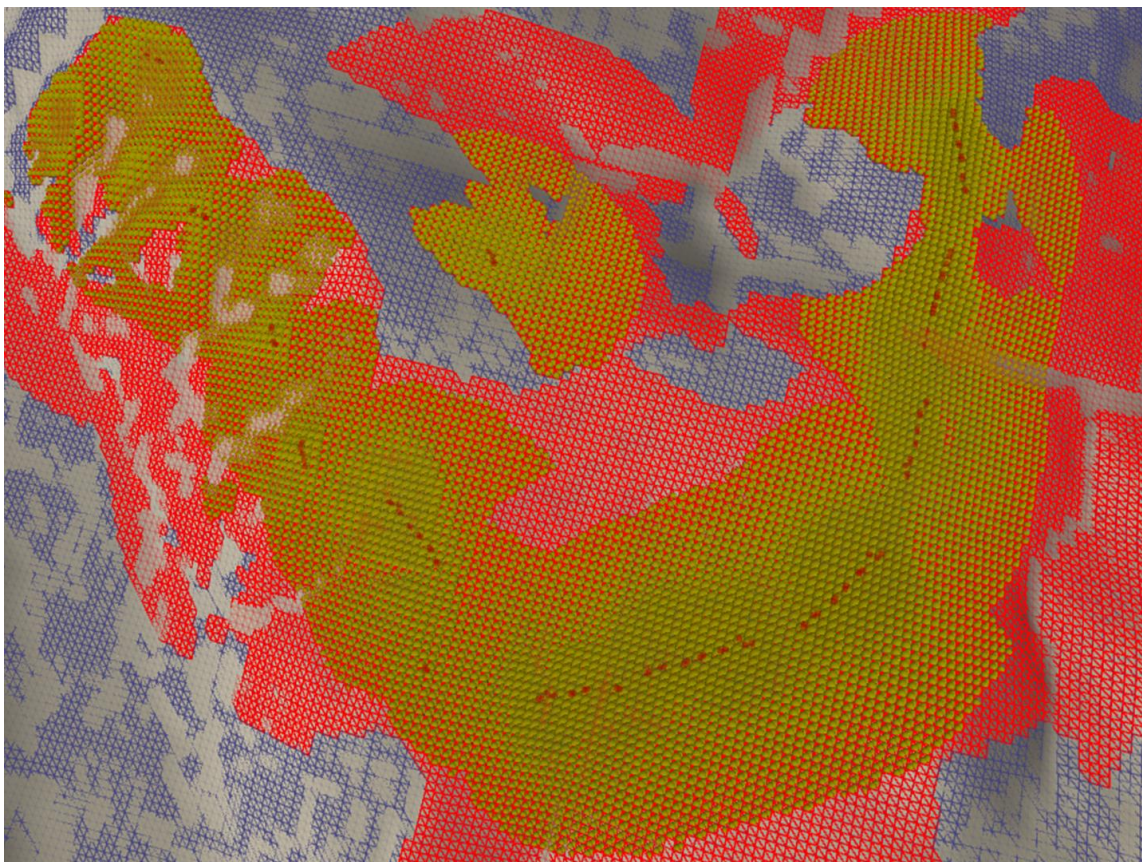


Figure 101. Scenario Group 6 Node Penalty Visualization

At the battalion level, the maneuver plan for Scenario Group 6 is a frontal attack. In the first stage, the first Friendly Force company conducts a frontal attack on the southernmost defensive positions. In the second stage, the other two Friendly Force companies attack the western and eastern flanks simultaneously. The western flank positions are generally assailed from behind, so the Friendly Force can attack downhill. The eastern flank is taken piece-by-piece, with a sequence of flank attacks from south to north. Due to the large area covered by this battalion attack, we show the assault directions in one image (Figure 102) and the starting positions and manual fire support positions in another (Figure 103). In the latter, the smaller formations on the northern portion of the Friendly Force are the machine gun squads.

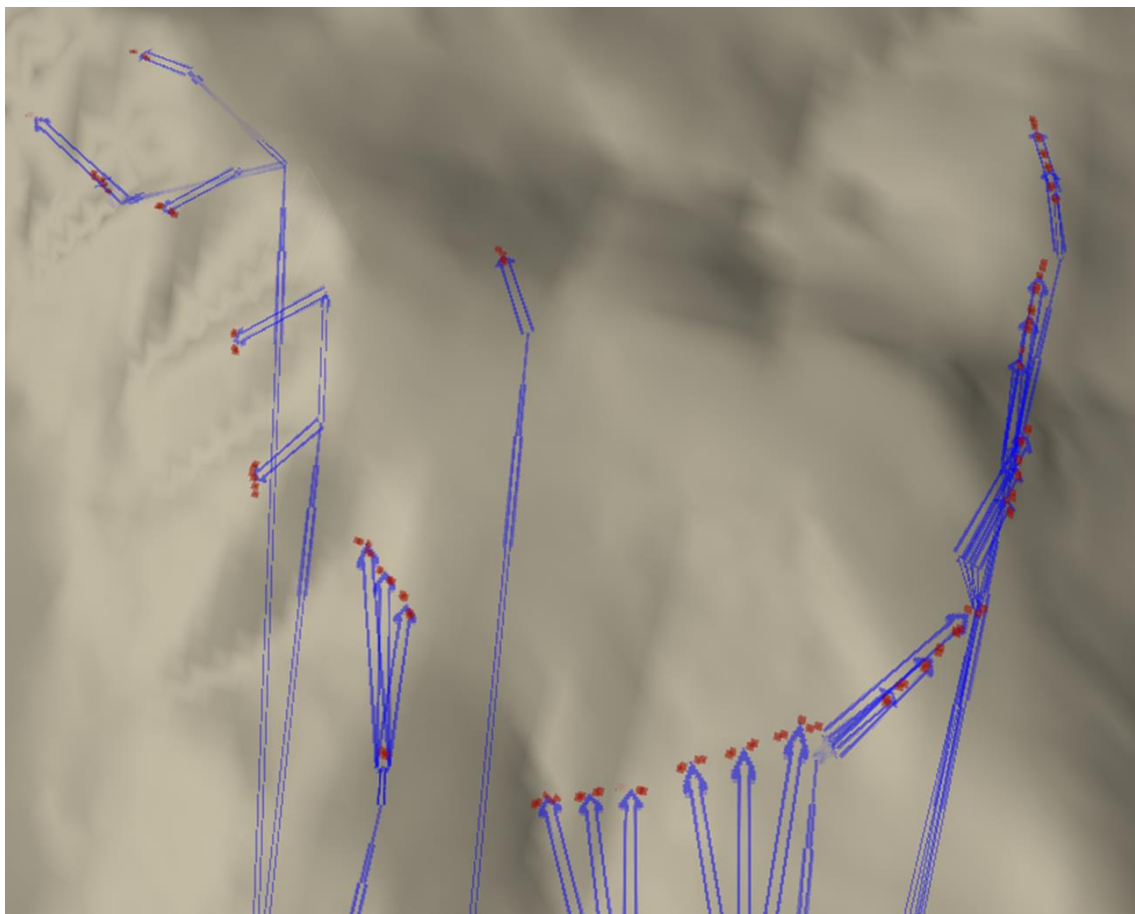


Figure 102. Scenario Group 6 Maneuver Plan Assaults

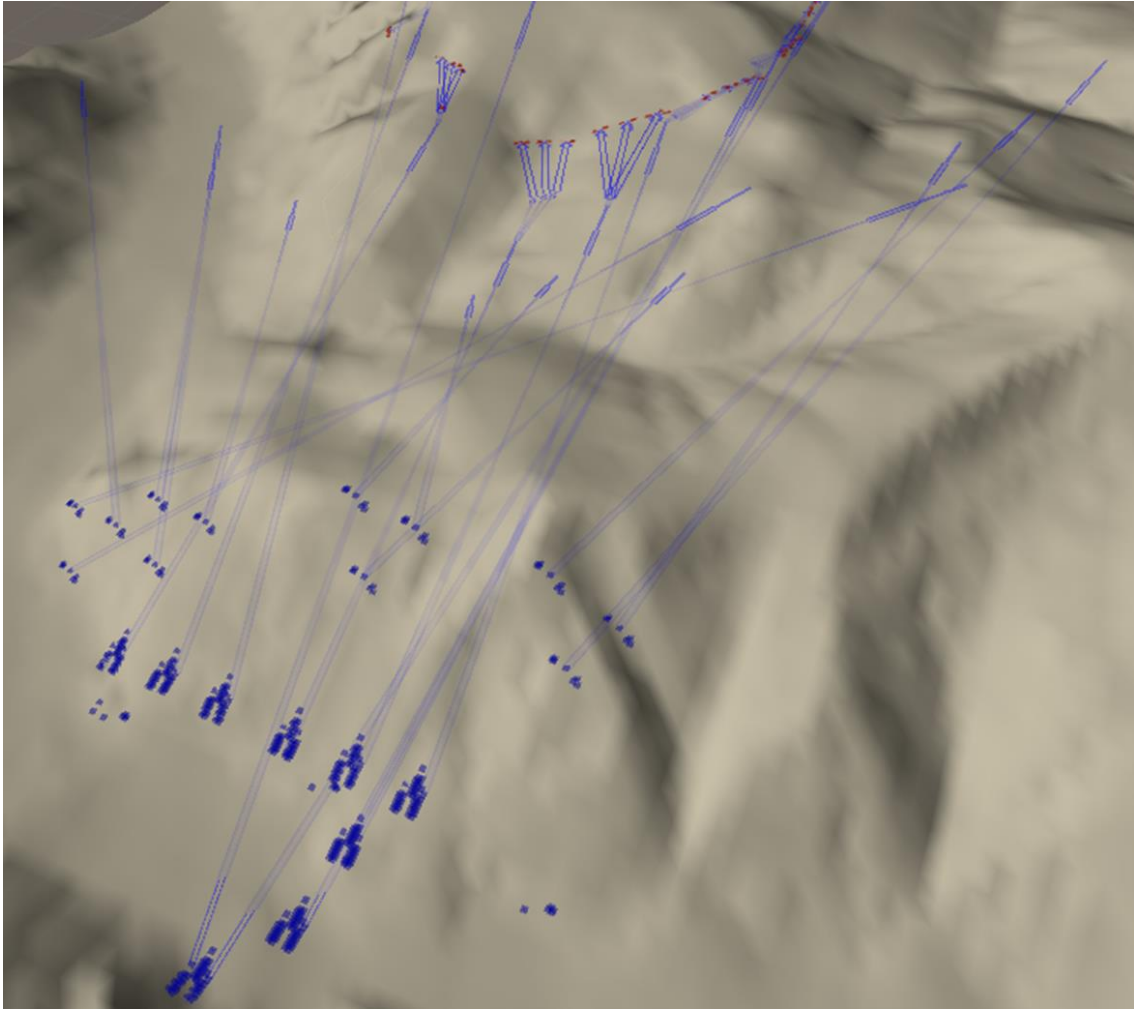


Figure 103. Scenario Group 6 Starting Positions and Manual Fire Support Positions

As usual, the close-quarters positions (Figure 104) include some with and some without safe paths to useful fire support positions. In one case, we have broken the normal rule that close-quarters positions may not be inside or behind the objective area. The offending Friendly Force unit is located west of the central Enemy Force unit. Since the attack plan goes through that area of the objective area, and since there is such a large unobserved space, the presence of a small unit inside enemy lines is more realistic. Although one would not send a machine gun squad on such a mission, deep reconnaissance is a well-known tactic. Placing a machine gun team here could be considered a crude representation of deep reconnaissance.

The front and most of the western flank of the Enemy Force's battle position offer little in the way of protected positions inside weapons range, so the machine gun squads are placed elsewhere. Note that, for Figure 104, we have doubled the size of the entity highlighting boxes to make them visible at this scale.

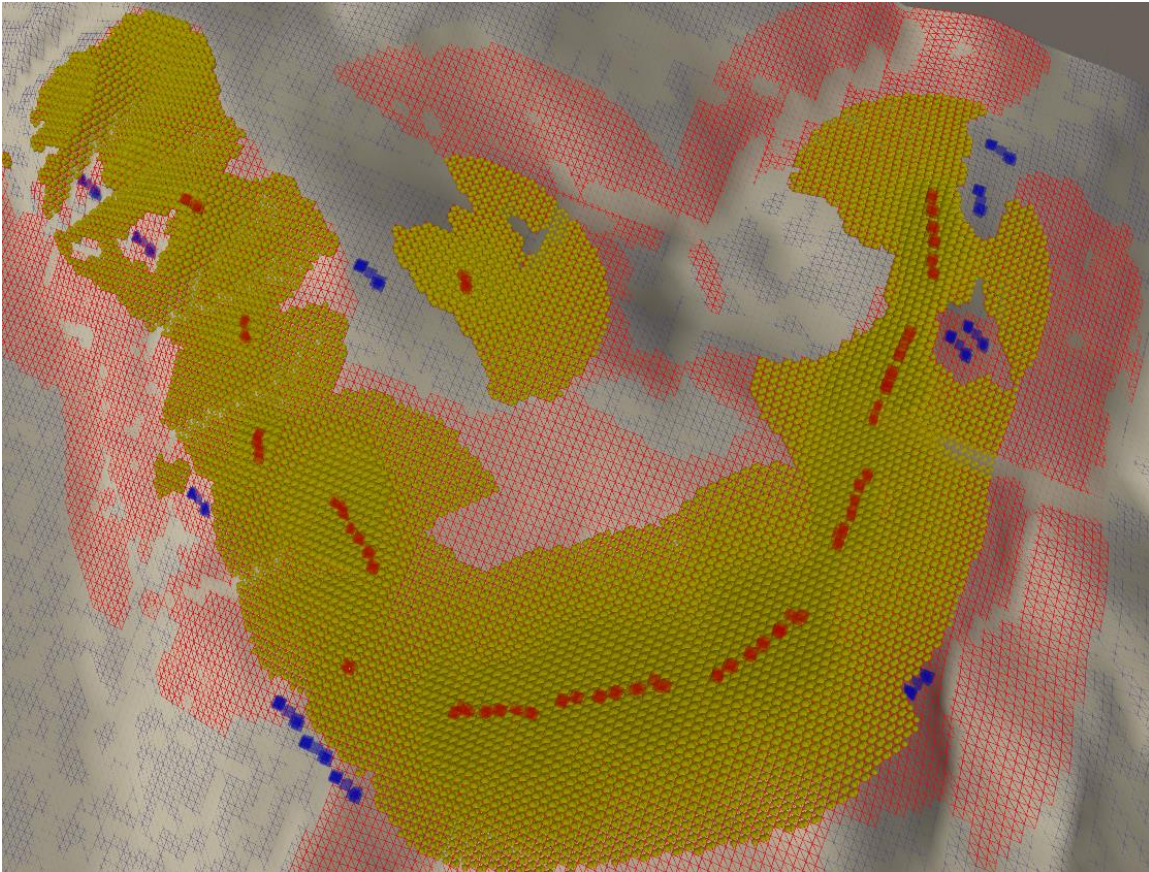


Figure 104. Scenario Group 6 Close-quarters Starting Positions

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

- Amstutz, Peter, Mitha Andra, and Daniel Rice. 2012. "Reasoning, Planning, and Goal Seeking: A Cognitive Approach for Small Combat Unit Constructive Simulation." In *Proceedings of the 2012 Symposium on Military Modeling and Simulation*, article 4. Society for Computer Simulation International.
- Ancker C. J., Jr., and A. V. Gafarian. 1988. *The Validity of Assumptions Underlying Current Uses of Lanchester Attrition Rates*. U.S. Army Training and Doctrine Command (TRADOC) Analysis Command (TRAC) Report No. TRAC-WSMR-TD-7-88. White Sands, NM: TRAC.
- Appleget, Jeffrey, Curtis Blais, Robert Burks, Richard F. Brown, Deborah Duong, Michael Jaye, Timothy Perkins, and Meredith Thompson. 2011. *Irregular Warfare Model Validation Best Practices Guide*. Monterey, CA: TRAC.
- Balogh, Imre, and G. Harless. 2003. "An Overview of the COMBAT XXI Simulation Model: A Model for the Analysis of Land and Amphibious Warfare." Presentation to the 71st Military Operations Research Society Symposium.
- Barnett, Correlli. 1963. *The Swordbearers: Studies in Supreme Command in the First World War*. London: Eyre & Spottiswoode.
- Billard, L. 1979. *Stochastic Lanchester-type Combat Models I*. NPS Report No. NPS55-79-022. Monterey, CA: NPS.
- Bohemia Interactive Simulations. 2015. *VBS3 User Manual*. Ver. 3.4.7. <https://manuals.bisimulations.com/vbs3/3-4/manuals/>.
- Box, George E. P., and Norman Richard Draper. 1987. *Empirical Model-Building and Response Surfaces*. New York: Wiley.
- Buro, Michael. 2003. "Real-Time Strategy Games: A New AI Research Challenge." In *Proceedings of The Joint Conference on Artificial Intelligence*, 1534–1535.
- Buss, Arnold. 2011. *Discrete Event Simulation Modeling*. Class notes. Monterey, CA: NPS.
- Chung, Michael, Michael Buro, and Jonathan Schaeffer. 2005. "Monte Carlo Planning in RTS Games". In *Proceedings of the Symposium on Computational Intelligence and Games*, 117–134.
- Churchill, David, and Michael Buro. 2013. "Portfolio Greedy Search and Simulation for Large-Scale Combat in Starcraft." In *Proceedings of the Conference on Computational Intelligence and Games*.

- Clark, Evan C., Joel C. Eichelberger, and Jeffrey A. Smith. 2010. "Tactical Behavior Composition." In *Proceedings of the 19th Conference On Behavior Representation In Modeling And Simulation*, 75-82. Charleston, SC: BRIMS Society.
- Darken, Christian J. 2004. "Visibility and Concealment Algorithms for 3D Simulations." In *Proceedings of the 13th Conference on Behavior Representation in Modeling and Simulation*.
- Darken, Christian, Daniel McCue, and Michael Guerrero. 2010. "Realistic Fireteam Movement In Urban Environments." In *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 9-14. Menlo Park, CA: AAAI Press.
- McIntosh, G. C. 2009. *MANA-V (Map Aware Non-Uniform Automata – Vector) Supplementary Manual*. Defence Technology Agency (DTA) Technical Note 2009/7, NR 1525, ISSN 1175-6608. Auckland, New Zealand: DTA.
- Department of Defense. 2016a. *Infantry Platoon And Squad*. Army Techniques Publication 3–21.8 (with Change 1). Washington, DC: Headquarters, Department of the Army, August 23.
- . 2016b. *Department Of Defense Dictionary Of Military And Associated Terms*. Joint Publication 1–02. Washington, DC, February 15.
- . 2015a. *COMBATXXI Users Guide*. White Sands, NM: TRAC. Available from <https://cxxi.wsmr.army.mil/confluence/display/CXXIDOC/Home>.
- . 2015b. *OneSAF Operators Manual*. OneSAF Control No. C001-OM-5.1. Orlando, FL: Program Executive Office, Simulation Training and Instrumentation, July 24.
- . 2014a. *Infantry Company Operations*. Marine Corps Warfighting Publication 3–11.1. Washington, DC: Deputy Commandant of the Marine Corps for Combat Development and Integration, October 6.
- . 2014b. *MTWS User Documentation*. Ver. 3.5.1. Orlando, FL: Program Manager, Training Systems, Marine Corps Systems Command, October 28.
- . 2013a. The DOD Modeling And Simulation Glossary. Web resource. <http://www.msco.mil/MSGlossary.html>.
- . 2013b. *Doctrine For The Armed Forces Of The United States*. Joint Publication 1. Washington, DC, March 25.
- . 2012a. *Offense And Defense*. Army Doctrinal Reference Publication 3–90. Washington, DC: Headquarters, Department of the Army, August 31.

- . 2012b. *The Operations Process*. Army Doctrine Reference Publication 5–0. Washington, DC: Headquarters, Department of the Army, May 17.
- . 2011a. *Joint Operations*. Joint Publication 3–0. Washington, DC, August 11.
- . 2011b. *Joint Operation Planning*. Joint Publication 5–0. Washington, DC, August 11.
- . 2010a. *Operational Terms and Graphics*. Army Field Manual 1–02 (FM 101–5-1) and Marine Corps Reference Publication 5–12A (With Change 1). Washington, DC: Headquarters, Department of the Army, and Headquarters, U.S. Marine Corps, February 2.
- . 2010b. *Marine Corps Planning Process*. Marine Corps Warfighting Publication 5–1. Washington, DC: Headquarters, U.S. Marine Corps, August 24.
- . 2009a. DOD Modeling and Simulation Verification, Validation, and Accreditation. DOD Instruction 5000.61. Washington, DC: Under Secretary of Defense for Acquisition, Technology, and Logistics, December 9.
- . 2009b. *Joint Intelligence Preparation of the Operational Environment*. Joint Publication 2–01.3. Washington, DC, June 16.
- . 2008. *Operations*. Army Field Manual 3–0. Washington, DC: Headquarters, Department of the Army, February 27.
- . 2003. *The SBCT Infantry Rifle Company*. Army Field Manual 3–21.11 (with Change 1). Washington, DC: Headquarters, Department of the Army, January 23.
- . 2001. *Recommended Practices Guide Special Topics: Validation of Human Behavior Representations*. Alexandria, VA: Modeling & Simulation Coordination Office, September 25.
- . 1997a. *Warfighting*. Marine Corps Doctrinal Publication 1. Washington, DC: Headquarters, U.S. Marine Corps, June 20.
- . 1997b. *Tactics*. Marine Corps Doctrinal Publication 1–3. Washington, DC: Headquarters, U.S. Marine Corps, July 30.
- Dijkstra, Edsger W. 1959. “A Note on Two Problems in Connexion with Graphs.” *Numerische Mathematik* 1 (1): 269–271.
- Donaldson, Michael J. 2014. *Modeling Dynamic Tactical Behaviors in COMBATXXI Using Hierarchical Task Networks*. Master of Science, Naval Postgraduate School.

- Doran, Jim E., and Donald Michie. 1966. "Experiments with the Graph Traverser Program." In *Proceedings of the Royal Society of London (A): Mathematical, Physical and Engineering Sciences*, 235–259. The Royal Society.
- Erol, K., J. Hendler, and D. S. Nau. 1996. "Complexity Results for HTN planning." *Annals of Mathematics and Artificial Intelligence* 18 (1): 78–84.
- Evertsz, Rick, Frank E. Ritter, Simon Russell, and David Shepherdson. 2007. "Modeling Rules of Engagement in Computer Generated Forces." In *Proceedings of Behavior Representation in Modeling and Simulation*.
- Fikes, Richard E., Peter E. Hart, and Nils J. Nilsson. 1972. "Learning and Executing Generalized Robot Plans." *Artificial Intelligence* 3: 251–288.
- Fikes, Richard E., and Nils J. Nilsson. 1971. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving." *Artificial Intelligence* 2 (3–4): 189–208.
- Ghallab, Malik, Dana Nau, and Paolo Traverso. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann Series in Artificial Intelligence. San Francisco, CA: Morgan Kaufmann Publishers.
- Gorniak, Peter, and Ian Davis. 2007. "SquadSmart: Hierarchical Planning and Coordinated Plan Execution for Squads of Characters." In *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 14–19. Menlo Park, CA: AAAI Press.
- Granberg, Aron. 2016. *The A* Pathfinding Project*. Ver. 3.8.2. Unity software asset. <http://arongranberg.com/astar/>.
- Gross, David C. 1999. *Report from the fidelity implementation study group*. Orlando, FL: Simulation Interoperability Standards Organization, SISO-REF-002-1999.
- Gu, Xueqiang, Jing Chen, Jie Li, and Haifeng Liu. 2012. "Genetic Vector Ordinal Optimization Algorithm Based on RSM for UCAV Attack Planning." In *Proceedings of the Second International Conference on Computer Science and Network Technology*, 1973–1977.
- Harder, Byron R., Imre Balogh, and Chris Darken. 2016. "Implementation of an Automated Fire Support Planner." In *Proceedings of the Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 51–57. Menlo Park, CA: AAAI Press.
- Harder, Byron R., and Chris Darken. 2016. "Automated Fire Support Planning for Combat Simulations." In *Proceedings of the International Conference on Social Computing, Behavioral-Cultural Modeling, & Prediction and Behavior Representation in Modeling and Simulation*. Washington, D.C.,

- Hayes, C. C., and J. L. Schlabach. 1998. "FOX-GA: A Planning Support Tool for Assisting Military Planners in a Dynamic and Uncertain Environment." *AAAI Technical Report WS-98-02*: 21-26.
- Helmbold, Robert L., and Aqeel A. Khan. 1986. *Combat History Analysis Study Effort (CHASE): Progress Report for the Period August 1984-June 1985*. CAA-TP-86-2. Bethesda, MD: U.S. Army Concepts Analysis Agency, August.
- Hill Jr., Randall W., Johnny Chen, Jonathan Gratch, Paul Rosenbloom, and Milind Tambe. 1998. "Soar-RWA: Planning, Teamwork, and Intelligent Behavior for Synthetic Rotary Wing Aircraft." In *Proceedings of the Seventh Conference on Computer Generated Forces & Behavioral Representation*, 177-188.
- Hinrichs, Thomas, Kenneth Forbus, Johan de Kleer, Sungwook Yoon, Eric Jones, Robert Hyland, and Jason Wilson. 2011. "Hybrid Qualitative Simulation of Military Operations." In *Proceedings of the Twenty-third Innovative Applications of Artificial Intelligence Conference*, 1655–1661. Menlo Park, CA: AAAI Press.
- Hoang, Hai, Stephen Lee-Urban, and Héctor Muñoz-Avila. 2005. "Hierarchical Plan Representations for Encoding Strategic Game AI." In *Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 63-68. Menlo Park, CA: AAAI Press.
- Hughes, Wayne P., Jr. 1994. "Uncertainty in Combat." *Military Operations Research* 1 (1): 45–57.
- . 1995. "Two Effects of Firepower: Attrition and Suppression." *Military Operations Research* 1 (3): 27–35.
- Ilachinski, Andrew. 2004. *Artificial War: Multiagent-based Simulation of Combat*. New Jersey: World Scientific Publishing Co. Pte. Ltd.
- Infinity Code. 2016. *Real World Terrain*. Ver. 2.5.0.7. Unity software asset. <http://www.infinity-code.com/en/products/real-world-terrain>.
- Isla, Damian. 2005. "Handling Complexity in the Halo 2 AI." In *Game Developers Conference Proceedings*. San Francisco, CA: UBM Technologies.
- Jurney, Chris. 2008. "Company of Heroes Squad Formations Explained." In *AI Game Programming Wisdom 4.*, edited by Steve Rabin. Boston, MA: Course Technology.
- Kabanza, Froduald, Philippe Bellefeuille, Francis Bisson, Abder Rezak Benaskeur, and Hengameh Irandoust. 2010. "Opponent Behaviour Recognition for Real-time Strategy Games." In *Proceedings of the Conference on Artificial Intelligence: Plan, Activity, and Intent Recognition Workshop*, 29–36. Menlo Park, CA: AAAI Press.

- Kelly, John-Paul, Adi Botea, and Sven Koenig. 2008. "Offline Planning with Hierarchical Task Networks in Video Games." In *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Menlo Park, CA: AAAI Press.
- Kewley, Robert H., and Mark J. Embrechts. 2002. "Computational Military Tactical Planning System." *IEEE Transactions on Systems, Man, and Cybernetics* 32 (2): 161–171.
- Laird, John E. 2012. *The SOAR Cognitive Architecture*. Cambridge, MA: MIT Press.
- Laird, John E., and Paul S. Rosenbloom. 2008. "Extending the SOAR Cognitive Architecture." In *Proceedings of the First Artificial General Intelligence Conference*, edited by Pei Wang, Ben Goertzel, and Stan Franklin, 224-235. Fairfax, VA: IOS Press.
- Lanchester, F. W. 1916. *Aircraft in Warfare: The Dawn of the Fourth Arm*. London: Constable and Company Limited.
- Lui, F., D. Jarvis, J. Jarvis, R. Connell, and J. Vaughan. 2002. "An Architecture to Support Autonomous Command Agents for OneSAF Testbed Simulations." In *Proceedings of the SimTecT Conference*. Rundle Mall, South Australia: Simulation Australasia, Ltd.
- Milkowski, Marcin. 2016. "The Computational Theory of Mind." In *The Internet Encyclopedia of Philosophy*. Accessed December 15. Available from <http://www.iep.utm.edu/compmind/>.
- Miller, David. 2016. *Hierarchical Task Network Prototyping in Unity3D*. Master's thesis, Naval Postgraduate School.
- Mononen, Mikko. 2010. *Simple Stupid Funnel Algorithm*. Video game development blog. Available from <http://digestingduck.blogspot.com/2010/03/simple-stupid-funnel-algorithm.html>.
- MOVES Institute. 2015. "Basic Observe, Communicate, Move, and Engage (BOCME) Hierarchical Task Network (HTN) Tutorial." User documentation for COMBATXXI Monterey Extensions. Monterey, CA: NPS.
- Muñoz-Avila, Héctor, and David W. Aha. 2004. "On the Role of Explanation for Hierarchical Case-based Planning in Real-time Strategy Games." In *Proceedings of the 7th European Conference on Case-Based Reasoning Workshop on Explanations in CBR*.
- Myers, Karen L. 1999. "A Continuous Planning and Execution Framework." *AI Magazine* 20 (4): 63–69.

- O'Connor, Dave. 2015. *Command Ops 2 Game Manual*. Lock 'n Load Publishing. Available from <http://www.panthergames.com/>.
- Ontañón, Santiago, and Michael Buro. 2015. "Adversarial Hierarchical-Task Network Planning for Complex Real-time Games." In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, 1652-1658. Menlo Park, CA: AAAI Press.
- Ontañón, Santiago, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. 2007. "Case-based Planning and Execution for Real-Time Strategy Games." In *Proceedings of the International Conference on Case-based Reasoning*. Belfast.
- Ontañón, Santiago, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. 2013. "A Survey of Real-time Strategy Game AI Research and Competition in StarCraft." In *IEEE Transactions on Computational Intelligence and AI in Games* 5 (4), 293-311.
- Orkin, Jeff. 2005. "Agent Architecture Considerations for Real-time Planning in Games." In *Proceedings of the First AAAI Artificial Intelligence and Interactive Digital Entertainment Conference*, 105-110. Menlo Park, CA: AAAI Press.
- Panther Games. 2015. *Command Ops 2*. Video game. Available from <http://www.panthergames.com/>.
- Petty, Mikel D. 2010. "Verification, Validation, and Accreditation." In *Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains*, edited by John A. Sokolowski by Catherine M. Banks, 325-372. Hoboken, NJ: Wiley.
- Pew, Richard W., and Anne M. Mavor, eds. 1998. *Modeling Human and Organizational Behavior: Application to Military Simulations*. Washington, D.C.: National Academy Press.
- Pittman, David. 2008. "Command Hierarchies Using Goal-Oriented Action Planning." In *AI Game Programming Wisdom 4*, edited by Steve Rabin, 383-391. Boston, MA: Course Technology.
- Ponsen, Marc, Héctor Muñoz-Avila, Pieter Spronck, and David W. Aha. 2006. "Automatically Generating Game Tactics Through Evolutionary Learning." *AI Magazine* 27 (3): 75-84.
- Reece, Douglas A. 2003. "Movement Behavior for Soldier Agents on a Virtual Battlefield." *Presence* 12 (4): 387-410.
- Reynolds, Craig. 1999. "Steering Behaviors for Autonomous Characters." In *Game Developers Conference Proceedings*. San Francisco, CA: UBM Technologies.

- Robertson, Glen, and Ian Watson. 2014. "A Review of Real-Time Strategy Game AI." *AI Magazine* 35 (4): 75–104.
- Rowe, Neil C. 2009. "Automated Instantaneous Performance Assessment for Marine-Squad Urban-terrain Training." In *Proceedings of the International Command and Control Research and Technology Symposium*.
- Rowe, Neil C., and David H. Lewis. 1989. "Vehicle Path-Planning in Three Dimensions Using Optics Analogs for Optimizing Visibility and Energy Cost." In *Proceedings of the NASA Conference on Space Telerobotics IV*, 217-226.
- Russell, Stuart J., and Peter Norvig. 2011. *Artificial Intelligence: A Modern Approach*. 3rd edition. Upper Saddle River, NJ: Pearson HE, Inc.
- Sacerdoti, Earl D. 1974. "Planning in a Hierarchy of Abstraction Spaces." *Artificial Intelligence* 5 (2): 115-135.
- Sanchez, Susan M. 2000. "Robust Design: Seeking the Best of All Possible Worlds." In *Winter Simulation Conference Proceedings*, 69–76.
- Schlabach, Jerry. 2010. Cognitive Amplification for Contextual Game-theoretic Analysis of Courses of Action Addressing Physical Engagements. U.S. patent 20100015579 A1, filed July 16, 2009, and issued January 21.
- Secarea V. V., Jr., and H. F. Krikorian. 1990. "Adaptive Multiple Target Attack Planning in Dynamically Changing Hostile Environments." In *Proceedings of the National Aerospace and Electronics Conference*, 1117-1123.
- Shi, Yinxuan, and Roger Crawfis. 2014. "Group Tactics Utilizing Suppression and Shelter." In *Proceedings of the International Conference on Computer Games: AI, Animation, Mobile, Interactive Multimedia, Educational and Serious Games*, 20-27.
- Silver, David. 2005. "Cooperative Pathfinding." In *Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 117–122. Menlo Park, CA: AAAI Press.
- Stanescu, Marius, Nicolas Barriga, and Michael Buro. 2015. "Using Lanchester Attrition Laws for Combat Prediction in StarCraft." In *Proceedings of the Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 86–92. Menlo Park, CA: AAAI Press.
- Stay At Home Devs. 2016. *Game Data Editor*. Ver. 2.6. Unity software asset. <http://stayathomedevs.com/>.
- Stout, Bryan. 2000. "The Basics of A* for Path Planning." In *Game Programming Gems.*, edited by Mark DeLoura. Boston, MA: Charles River Media.

- Straatman, Remco, William van der Sterren, and Arjen Beij. 2005. "Killzone's AI: Dynamic Procedural Combat Tactics." In *Game Developers Conference Proceedings*. San Francisco, CA: UBM Technologies.
- Straatman, Remco, Tim Verweij, Alex Champandard, Robert Morcus, and Hylke Kleve. 2013. "Hierarchical AI for Multiplayer Bots in Killzone 3." In *Game AI Pro: Collected Wisdom of Game AI Professionals*, edited by Steve Rabin, 377–390. Boca Raton, FL: CRC Press.
- Temiz, Yusuf Ziya. 2016. *Building a 3D Agent-based Simulation of Artillery Survivability*. Master's thesis, Naval Postgraduate School.
- Tozour, Paul. 2003. "Search Space Representations." In *AI Game Programming Wisdom 2*, edited by Steve Rabin. Brookline, MA: Charles River Media.
- Unity Technologies. 2016. *Unity*. Video game development software. Ver. 5.4.2f2. Available from <https://unity3d.com/>.
- Uriarte, Alberto, and Santiago Ontañón. 2015. "Automatic Learning of Combat Models for RTS Games." In *Proceedings of the Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 212–218. Menlo Park, CA: AAAI Press.
- van der Sterren, William. 2014. *PlannedAssault*. Website and software application. <http://www.plannedassault.com>.
- . 2013. "Hierarchical Plan-Space Planning for Multi-unit Combat Maneuvers." In *Game AI Pro: Collected Wisdom of Game AI Professionals*, edited by Steven Rabin, 169–183. Boca Raton, FL: CRC Press.
- . 2002. "Tactical Path-finding with A*." In *Game Programming Gems 3*, edited by Mark DeLoura, 294–306. Boston, MA: Course Technology.
- Wang, Che, Pan Chen, Yuanda Li, Christoffer Holmgard, and Julian Togelius. 2016. "Portfolio Online Evolution in StarCraft." In *Proceedings of the Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Menlo Park, CA: AAAI Press.
- Wang, Xing, Stephen George, Jason Lin, and Jyh-Charn Liu. 2015. "Quantifying Tactical Risk: A Framework for Statistical Classification Using MECH." In *Proceedings of the 8th International Conference of Social Computing, Behavioral-Cultural Modeling, and Prediction*, 446–451. New York: Springer.
- Washburn, Alan, and Moshe Kress. 2009. *Combat Modeling*. International Series in Operations Research and Management Science. New York: Springer.

Wilkins, David E., and Roberto V. Desimone. 1994. "Applying an AI Planner to Military Operations Planning." In *Intelligent Scheduling*, edited by Mark Fox and Monte Zweben, 685. San Francisco, CA: Morgan Kaufmann Publishers, Inc.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California